

## Awesome LLM Inference v0.3

LITE: Accelerating LLM Inference by Enabling Intermediate Layer Decoding -----	1
AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration -----	21
ByteTransformer: A High-Performance Transformer Boosted for Variable-Length Inputs -----	34
Orca: A Distributed Serving System for Transformer-Based Generative Models -----	46
Efficient LLM Inference on CPUs -----	65
FastServe: Fast Distributed Inference Serving for Large Language Models -----	71
Flash-Decoding for long-context inference -----	85
FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness -----	90
FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning -----	124
FLASHDECODING++: FASTER LARGE LANGUAGE MODEL INFERENCE ON GPUS -----	138
FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU -----	154
Flexible Performant GEMM Kernels on GPUs -----	177
FP8-LM: Training FP8 Large Language Models -----	195
FP8 FORMATS FOR DEEP LEARNING -----	218
LibShalom: Optimizing Small and Irregular-shaped Matrix Multiplications on ARMv8 Multi-Cores -----	227
LLM-FP4: 4-Bit Floating-Point Quantized Transformers -----	240
SHEARED LLAMA: ACCELERATING LANGUAGE MODEL PRE-TRAINING VIA STRUCTURED PRUNING -----	254
LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale -----	274
Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism -----	294
NVIDIA Tensor Core Programmability, Performance & Precision -----	309
Online normalizer calculation for softmax -----	321
PagedAttention: Efficient Memory Management for Large Language Model Serving with PagedAttention -	330
Reducing shared memory footprint to leverage high throughput on Tensor Cores and its flexible API extension library -----	346
ROFORMER: ENHANCED TRANSFORMER WITH ROTARY POSITION EMBEDDING -----	355
SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models -----	369
SpecInfer: Accelerating Generative Large Language Model Serving with Speculative Inference and Token Tree Verification -----	382
SpQR: A Sparse-Quantized Representation for Near-Lossless LLM Weight Compression -----	397
StreamLLM: EFFICIENT STREAMING LANGUAGE MODELS WITH ATTENTION SINKS -----	423
WINT8: Who Says Elephants Can't Run: Bringing Large Scale MoE Models into Cloud Scale Production -	435
ZeroQuant: Efficient and Affordable Post-Training Quantization for Large-Scale Transformers -----	443
ZeroQuant-FP: A Leap Forward in LLMs Post-Training W4A8 Quantization Using Floating-Point Formats -----	467
ZeroQuant-V2: Exploring Post-training Quantization in LLMs from Comprehensive Study to Low Rank Compensation -----	477

---

# Accelerating LLM Inference by Enabling Intermediate Layer Decoding

---

Neeraj Varshney    Agneet Chatterjee    Mihir Parmar    Chitta Baral  
Arizona State University

## Abstract

Large Language Models (LLMs) have achieved remarkable performance across a wide variety of natural language tasks; however, their large size makes their inference slow and computationally expensive which poses a practical challenge for resource constrained real-world applications. Focusing on this problem, we propose to instruction tune LLMs in a way that enables intermediate layer decoding for efficiently generating text, but importantly without compromising the quality of the generation. Specifically, we instruction tune LLMs with additional explicit Losses from the InTermediate layErs (LITE) and show that it enables these layers to acquire ‘good’ generation ability without affecting the generation ability of the final layer. We perform ‘*dynamic confidence-based early exiting*’ at token level from the intermediate layers which improves the efficiency of inference while maintaining the generation quality. We conduct comprehensive experiments by instruction tuning LLaMA-2 models on the widely used Alpaca dataset and holistically evaluate on four different human-instruction test sets: Vicuna, WizardLM, Koala, and Self-Instruct. We show that **dynamic early exiting achieves consistent and considerable cost improvements (37.86% on average) while maintaining the generation quality of the responses.** We further conduct a thorough analysis of the results over several important aspects, such as comparing the semantic similarity of the outputs and dissecting the efficiency improvements by comparing the number of tokens generated in the output. In summary, our work contributes to improving the efficiency of LLM inference while maintaining the generation quality, a crucial step en route to enabling their widespread adoption.

## 1 Introduction

Recently developed Large Language Models (LLMs) [32, 3, 8, 28, 30] have revolutionized the field of Natural Language Processing and achieved remarkable performance across a wide variety of tasks ranging from text generation and question answering to code generation and complex reasoning. ‘Instruction Tuning’ further teaches these language models to follow the user’s instruction provided in natural language [40, 25, 29, 38, 9]. Despite all the notable abilities of these models, their large size (number of parameters) makes their inference slow and computationally expensive which poses a practical challenge limiting their widespread adoption in resource constrained real-world applications. Focusing on the above problem, in this work, **we propose to instruction tune LLMs in a way that enables intermediate layer decoding for efficiently generating text without compromising the quality of the generation.**

We first show that in standard instruction tuning, only the final layer of the model acquires the ability to generate ‘*quality*’ text while the representations of the intermediate layers (when passed through the language modeling head) fail to do so. This restricts decoding from these intermediate layers without degrading the generation quality. Addressing this point, we propose to instruction tune LLMs with additional explicit Losses from the InTermediate layErs (LITE) and show that it enables these layers to acquire ‘good’ generation ability. Importantly, we show that these layers acquire this ability without affecting the generation ability of the final layer; however, as expected, their generation ability still remains slightly inferior to the generation ability of the final layer. Thus, decoding the

complete response from an intermediate layer improves the efficiency of inference but still results in minor degradation in the quality of the response.

Addressing the above limitation, we show that (a) LITE greatly aligns the intermediate layers’ token prediction with that of the final layer and (b) the intermediate layers’ token prediction probabilities provide a strong signal of this alignment. Building on top of these two findings, we perform ‘**dynamic confidence-based early exiting**’ at token level from the intermediate layers which improves the efficiency of inference while maintaining the generation quality.

We conduct comprehensive experiments by instruction tuning LLaMA-2 models [32] on the widely used Alpaca dataset [31] and holistically evaluate on four different human-instruction test sets including Vicuna [7], WizardLM [44], Koala [13], and Self-Instruct [37]. Figure 1 compares the **quality of responses** (evaluated using the Claude model as detailed in Section 4) and the **inference cost** (measured in FLOPs) of the (i) standard generation method from the final layer with (ii) our dynamic early exiting method. It shows that **dynamic early exiting achieves consistent and considerable cost improvements (37.86% on average)** while maintaining the generation quality.

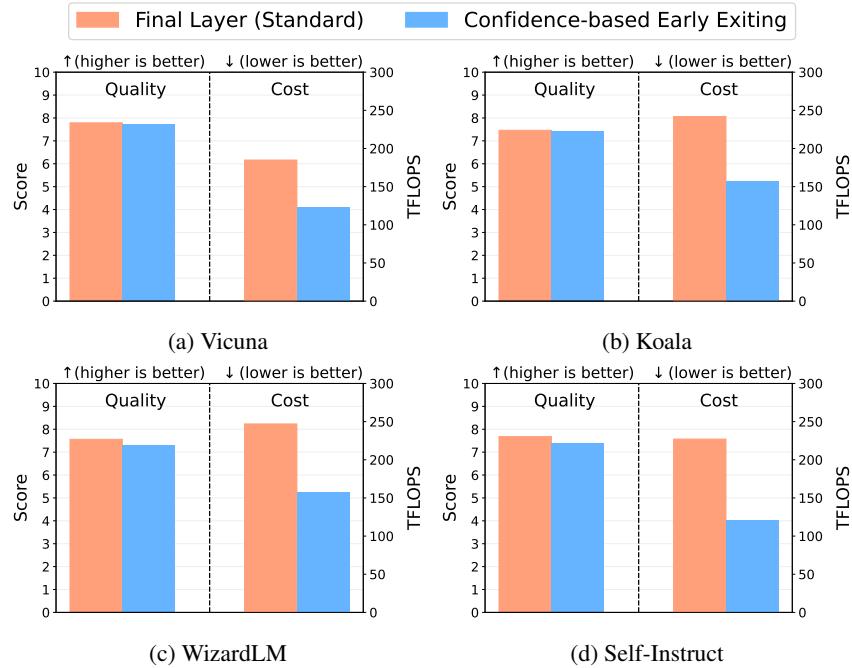


Figure 1: Comparing the quality of responses (evaluated using the Claude model) and the inference cost (measured in FLOPs) of the standard generation from the final layer with our dynamic early exiting method. It shows that **the dynamic early exiting achieves consistent and considerable cost improvements (37.86% on average) without degrading the generation quality**. For this study, the LLaMA-2 models are instruction tuned on the Alpaca dataset and evaluated on four test sets, namely, Vicuna, Koala, Self-Instruct, and WizardLM.

We further perform a thorough analysis of the results by (a) studying the quality and inference cost comparison at a category level on the evaluation datasets (5.5.1), (b) analyzing the semantic similarity between the responses generated from the final layer and the early exiting method (5.5.2), and (c) dissecting the efficiency improvements of the dynamic early exiting method by comparing the number of tokens generated in the outputs (5.5.3).

In summary, we show that instruction tuning with additional explicit losses from the intermediate layers (LITE) enables the intermediate layers to acquire ‘good’ generation ability and our ‘dynamic early exiting’ method leverages that to improve the efficiency of inference while maintaining the generation quality. We further discuss the potential of intermediate layer decoding in ‘speculative sampling’ and ‘hallucination detection’.

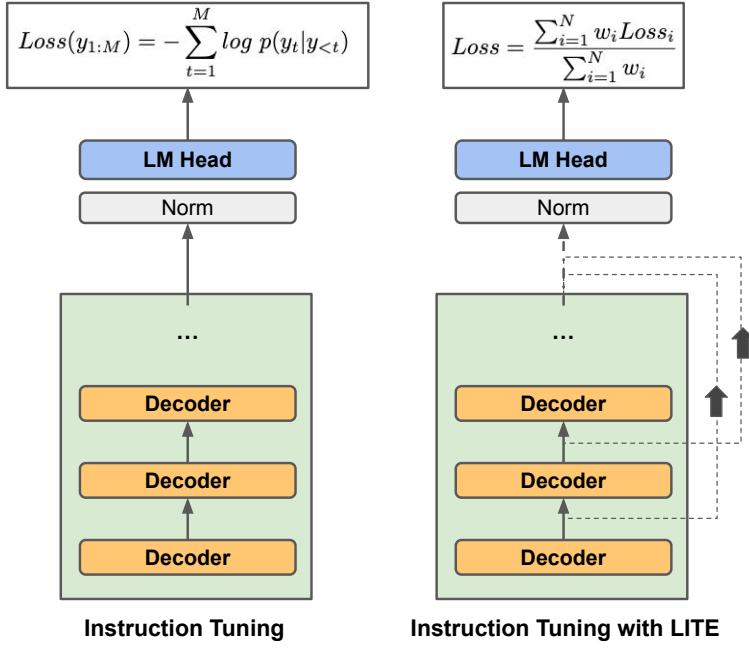


Figure 2: Illustrating loss calculation for standard instruction tuning (left) and instruction tuning with additional explicit losses from the intermediate layers LITE (right).

## 2 Instruction Tuning with Losses from Intermediate Layers

In this section, we first detail the standard instruction tuning (IT) and then describe instruction tuning with additional explicit losses from the intermediate layers (IT with LITE).

**Instruction Tuning (IT):** One of the major reasons that necessitate instruction tuning of LLMs is the mismatch between their pre-training objective and the users’ objective, i.e., LLMs are typically trained on minimizing the word prediction error on large corpora; while users want the model to follow their instructions. To this end, an instruction tuning dataset typically consisting of (instruction, input, and output) triplets is collected and a pre-trained model is fine-tuned in a fully supervised manner, where given the instruction and the input, the model is trained by predicting each token in the output [25, 9, 41, 38, 27]. Loss calculation during instruction tuning of a typical decoder-only LLM (LLaMA in this case) is shown in Figure 2 (left). The model consists of a stack of decoder layers followed by a language modeling head which outputs the probability distribution over the vocabulary tokens as its prediction. During the supervised fine-tuning, the loss over the output tokens is backpropagated from the final layer of the model:

$$Loss(y_{1:M}) = -\sum_{t=1}^M \log p(y_t|y_{<t})$$

**Instruction Tuning with Losses from Intermediate Layers (IT with LITE):** We show that in the standard instruction tuning, only the final layer of the model acquires the ability to generate ‘quality’ text while the representations of the intermediate layers (when passed through the language modeling head) fail to do so (Section 5.1). In other words, it does not explicitly teach the intermediate layers of the tuned LLM to generate tokens. This restricts decoding from these intermediate layers without degrading the generation quality.

We note that during tuning, the same language modeling head (that is used with the final layer) can also be used with the intermediate layers to obtain the losses of those layers. Note that this does not impact the number of parameters of the model as the same shared language modeling head is used for obtaining these losses. To this end, we calculate a weighted aggregation of the losses from the intermediate layers (including the final) to calculate the overall loss value as shown below:

$$Loss = \frac{\sum_{i=1}^N w_i Loss_i}{\sum_{i=1}^N w_i}$$

where  $N$  is the number of layers,  $w_i$  is the weight corresponding to the  $i^{th}$  layer, and  $Loss_i$  is the cross entropy loss of the  $i^{th}$  layer as shown in Figure 2.

During tuning, we use the representations of the intermediate layers and calculate the loss from these layers at the end. We note that this is a general formulation as it captures a variety of scenarios including the standard fine-tuning in which the loss is calculated only from the last layer (i.e.,  $w_i = 0$  for  $i=1$  to  $N-1$  and  $w_N = 1$ ). Furthermore, this formulation also allows aggregating losses from only the selected intermediate layers instead of all the layers by accordingly defining the LM head pathways and the  $w_i$  values. In Section 5.2, we will show that IT with LITE while enabling the intermediate layers with ‘good’ generation ability does not adversely affect the final layer’s generation ability. Furthermore, as expected, the quality of generation typically improves with the layer number as the later layers have more capacity to learn.

### 3 Making Inference Efficient

In this section, we first detail auto-regressive inference and then describe early exiting techniques, namely, fixed early exiting (Section 3.1) and dynamic confidence-based early exiting (Section 3.2).

**Auto-Regressive Inference:** In the context of language models, auto-regressive inference refers to the process of generating sequence of tokens where each token is generated based on the preceding tokens in the sequence. For generating a token, the model takes the input (including the previously generated tokens) and runs a forward pass in which the input is fed to the model and passed sequentially along its layers until the probabilities for the next token are predicted (called as **logits**). Chaining model forward passes with next token selection iteratively leads to the generation of text.

Auto-regressive language generation is based on the assumption that the probability distribution of a token sequence can be decomposed into the product of conditional next token distributions:

$$P(w_{1:T}|W_0) = \prod_{t=1}^T P(w_t|w_{1:t-1}, W_0)$$

where  $W_0$  corresponds to the given input prompt and the length  $T$  is usually determined on the fly when the token EOS is generated or when the specified maximum number of tokens have been generated. In **greedy decoding**, the token with the highest probability is selected as the next word prediction at each timestep  $t$ .

$$w_t = \operatorname{argmax}_w P(w|w_{1:t-1}, W_0)$$

During sampling, a cache of the keys and values can be maintained for every attention layer (called KV caching) which reduces the computations required at inference time. However, this increases the GPU VRAM memory requirement of inference.

In the following subsections, we describe early exiting techniques for efficient inference.

#### 3.1 Fixed Early Exiting

Since instruction tuning with LITE enables the intermediate layers to acquire ‘good’ generation ability, the computations during inference can be terminated at a pre-specified intermediate layer (referred to as **exiting layer**) and the language modeling head can be used to predict the next token. This saves the computations of the remaining layers that follow the specified exiting layer and thus it improves the efficiency of inference.

Though this method of fixed early exiting leads to improvement in the efficiency of inference, it is bound to result in some degradation in the quality of the generation as the generation ability of an intermediate layer still remains inferior to the generation ability of the final layer. We also note that the quality of generation typically improves with the layer number as the later layers have more capacity and hence ability.

### 3.2 Dynamic Confidence-Based Early Exiting

Addressing the limitation of the fixed early exiting method, we study a dynamic early exiting method that decides the exiting layer for a token prediction based on the intermediate layer’s probability of the prediction (softmax over the logit values).

This is motivated by our following two findings:

- (a) Instruction Tuning with LITE greatly aligns the intermediate layers’ token prediction with that of the final layer (Section 5.3) and
- (b) The intermediate layers’ token prediction probabilities (referred to as **confidence**) provide a strong signal of this alignment (Section 5.4).

Building on top of these two findings, we perform ‘*dynamic confidence-based early exiting*’ at token level from the intermediate layers which improves the efficiency of inference while maintaining the generation quality. Specifically, a set of intermediate layers with their corresponding confidence thresholds are defined and at inference time, the exiting decision for a prediction is taken by comparing the intermediate layer’s prediction confidence against its corresponding threshold. This enables the model to do efficient inference without degrading the quality of generation.

Note that we explore this exiting method for inference without using KV caching. This is because the standard KV caching cannot be used here as the representations of the layers after the exiting layer are not computed in this method and thus will not be available in the cache for the next token prediction if the model exits from a higher layer than the previous token prediction.

## 4 Experimental Setup

**Tuning and Inference:** We instruction tune the LLaMA-2 models [32] (7B) with the widely used Alpaca dataset [31]. Alpaca consists of 52K instruction-following demonstrations generated from OpenAI’s text-davinci-003 using the self-instruct [37] technique. For IT with LITE, we aggregate losses from the following selected intermediate layers: (8, 12, 16, 20, 24, 28) along with the final layer and use equal weights in loss calculation. We do full parameter fine-tuning on 4 A100 GPUs. We skip selecting the initial layers because they have a limited capacity to learn and do not result in good token predictions. We train this model for 5 epochs so that it achieves training loss comparable to standard tuning. For inference, we set the max new tokens value to 256.

We present all the results in this work corresponding to this tuning configuration and leave the exploration of selecting different intermediate layers and different weights for these layers for future work.

**Evaluation Datasets:** To perform holistic evaluation, we experiment with four different human-instruction test sets including Vicuna [7], Self-Instruct [37], Koala [13], and WizardLM [44]. We select these evaluation test sets as they can together cover a large number and types of instructions thus resulting in a comprehensive evaluation. Table 1 shows the statistics of the evaluation datasets.

Test Set	# Samples
Vicuna	80
Koala	180
WizardLM	218
Self Instruct	252

Table 1: Statistics of evaluation datasets.

**Evaluation Methodology:** The evaluation of the instruction-following ability of LLMs is challenging due to the existence of multiple correct responses to an input and the infeasibility of reproducing human evaluations. Addressing this problem, recent works have started to rely on automatic evaluations using LLMs [48, 7]. Specifically, a large language model like GPT-4 [26] or Claude [2] is used as a judge to compare the quality of responses of two models on a given instruction.

We note that these LLMs have been shown to be vulnerable to position bias in their judgment [36]. To circumvent this bias, we evaluate a response pair with both the orderings of the responses and then aggregate the judgment scores.

We provide the prompt for comparing the quality of the responses of two models in Appendix A. We call the Claude API with temperature set to 0 with ‘anthropic.claude-v1’ model.

## 5 Results

In this section, we first demonstrate the inability of the intermediate layers of the model tuned with standard instruction tuning to generate ‘quality’ text (5.1). Then, we proceed to show the impact of instruction tuning with LITE. Specifically, we first show that instruction tuning with LITE does not adversely affect the generation quality of the final layer (5.2). Then, we show that LITE aligns the intermediate layers’ token predictions with the final layer (5.3), and the corresponding prediction confidence values provide a strong signal of this alignment (5.4). These two findings motivate dynamic confidence-based early exiting method. Finally, we show the effectiveness of early exiting in improving the efficiency of inference while maintaining the generation quality (5.5).

### 5.1 Inability of the Intermediate Layers of the Model Tuned with Standard Instruction Tuning to Generate ‘High-Quality’ Text

In order to obtain the text (sequence of tokens) generated via fixed exiting from an intermediate layer, we apply the normalization (RMSNorm) followed by the language modeling head to the representations of that intermediate layer and skip the computations of the layers following the exiting layer (as detailed in Section 3.1). For the model tuned with the standard instruction tuning, we compare the quality of the text generated from different intermediate layers against the final layer’s generation in Figure 3. We compare the quality using the Claude model as detailed in Section 4. As expected, the intermediate layers generate text of considerably degraded quality and this quality drops as the layer number decreases.

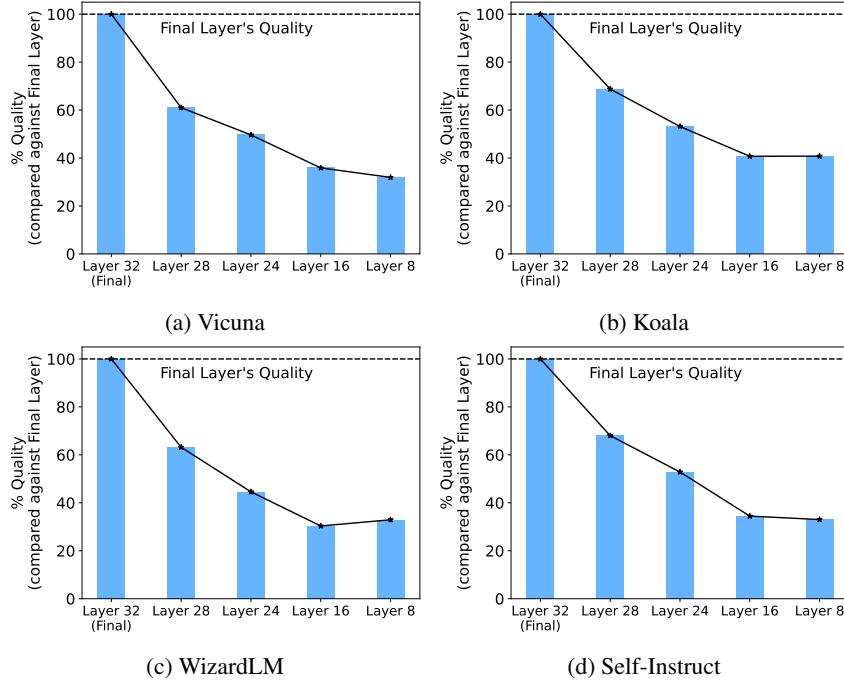


Figure 3: Demonstrating quality comparison of the text generated from different intermediate layers (via fixed exiting) against the final layer’s generation of the model tuned with standard instruction tuning. It shows that **the intermediate layers generate text of considerably degraded quality and this quality drops as the layer number decreases**.

This demonstrates that with standard instruction tuning, only the later layers (especially the final layer) of the model acquires the ability to generate ‘quality’ text while the representations of the intermediate layers (when passed through the language modeling head) fail to do so.

Thus, for the model tuned with standard instruction tuning, the early exiting inference method saves the inference computation cost but considerably degrades the quality of the generation. This restricts employing such early exiting techniques for the model tuned with standard instruction tuning. We show examples of responses obtained via fixed early exiting from different intermediate layers of the model in Appendix B.1.

We perform instruction tuning with LITE to enable the intermediate layers to acquire ‘good’ generation ability. Importantly, we note that these layers acquire this ability without affecting the generation ability of the final layer as we will show in the next subsection.

## 5.2 IT with LITE Does NOT Adversely Affect the Final Layer’s Generation Ability

In Figure 4, we compare the quality of responses (judged on a scale of 1 to 10 by the Claude Model) of (a) the model tuned using standard instruction tuning (IT) and (b) the model tuned using IT with LITE. Note that the responses for both these models correspond to their respective final layer’s output.

From the figure, it can be observed that for all the datasets, **the outputs of both models are of comparable quality which shows that tuning with LITE does not adversely affect the generation ability of the final layer of the model.**

Next, we demonstrate two important characteristics of instruction tuning with LITE (in 5.3 and 5.4) that motivate us to study dynamic confidence-based early exiting from the intermediate layers.

## 5.3 IT with LITE

### Greatly ‘Aligns’ Intermediate Layer Token Predictions with that of the Final Layer

We define percentage ‘alignment’ of a layer as the measurement of how often the token predictions of that layer match with the token predictions of the final layer (given equal prefixes).

For this study, we do not do early exiting, instead we just use the representation of each intermediate layer and pass it through the LM head to obtain the corresponding token prediction of each layer. Note that for generating the next token, we follow the standard generation methodology and append the predicted token of the last layer to the input. Essentially, we obtain the token prediction of all the layers given the same prefix.

In Figure 5, we plot the percentage alignment of token predictions of all intermediate layers with the token predictions of the final layer. The figure shows the percentage alignment of (i) the model tuned using standard IT (orange) and (ii) the model tuned using IT with LITE (blue). We show this result aggregated over all the output token predictions for all the inputs of the corresponding dataset.

We draw the following inferences from this figure:

- (a) **The predictions of the intermediate layers of the model tuned with LITE align well with the final layer**, i.e., given a prefix for this model, the intermediate layers’ token predictions match quite well with the final layer’s token prediction.
- (b) In contrast for the model tuned using IT, the token predictions of the intermediate layers do not align that well with the token prediction of its final layer.
- (c) As the layer number increases, the percentage alignment also increases, i.e., **given a prefix, the predicted token of the later layers show higher alignment (with predicted token of the final layer) than the initial layers.**

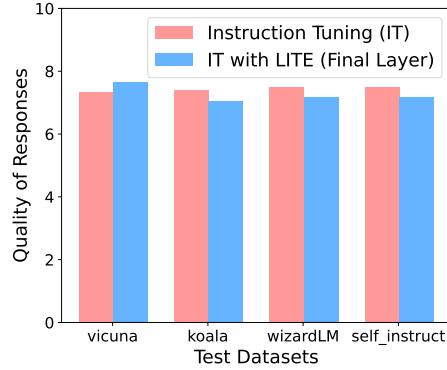


Figure 4: Comparing the quality of responses (judged on a scale of 1 to 10 by the Claude Model) of (a) the model tuned using standard instruction tuning (IT) and (b) the model tuned using IT with LITE. The outputs of the models are of comparable quality which highlights that **instruction tuning with LITE does not adversely affect the generation quality of the final layer of the model.**

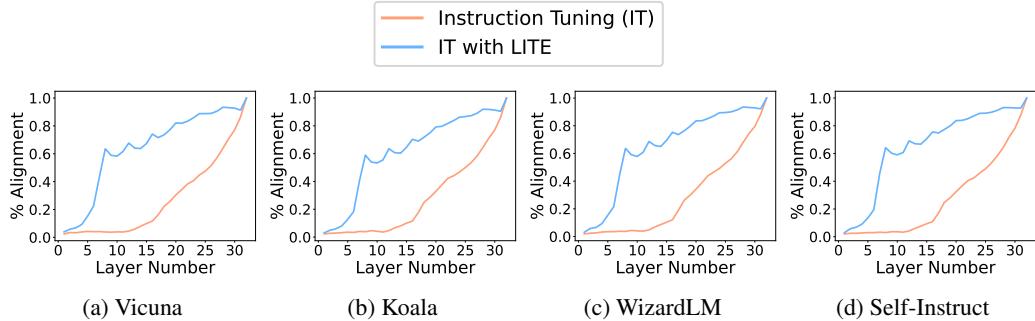


Figure 5: Comparing percentage ‘alignment’ of intermediate layer token predictions with the token predictions of the final layer for the model tuned using IT (orange) and the model tuned using IT with LITE (blue). This result is aggregated over all the output token predictions for all the inputs of the corresponding dataset.

(d) There are **some peaks in the blue curve** (IT with LITE) which correspond to the selected layers from which the intermediate loss is aggregated during tuning, i.e., these layers show higher alignment. Conversely, layers from which the loss is not aggregated during tuning tend to show slightly lesser alignment.

In summary, this study demonstrates that IT with LITE greatly aligns the token predictions of intermediate layers with those of the final layer.

In the next subsection, we show that the token prediction probabilities of the intermediate layers provide a strong signal of this alignment, i.e., when the probability is high the token prediction of an intermediate layer is more likely to align with the token prediction of the final layer. These two findings motivate the dynamic confidence-based early exiting.

#### 5.4 In IT with LITE, Intermediate Layers' Token Prediction Probabilities Indicate Likelihood of Alignment with the Final Layer's Token Prediction

Here, we plot the relationship between the token prediction confidence (softmax over the logits of the LM head) of the intermediate layers and the percentage alignment with the token prediction of the final layer. Figure 6 shows this plot for the model tuned with LITE. The figure shows that in IT with LITE, the intermediate layers’ token prediction probabilities provide a strong signal of alignment, i.e., a high token prediction confidence implies a higher likelihood of its alignment with the token prediction of the final layer. It also shows that with the increase in the layer number, the percentage alignment typically increases at the same confidence values.

In contrast, in standard instruction tuning (IT), the confidence is not well correlated with the percentage alignment as we show in Appendix B.3.

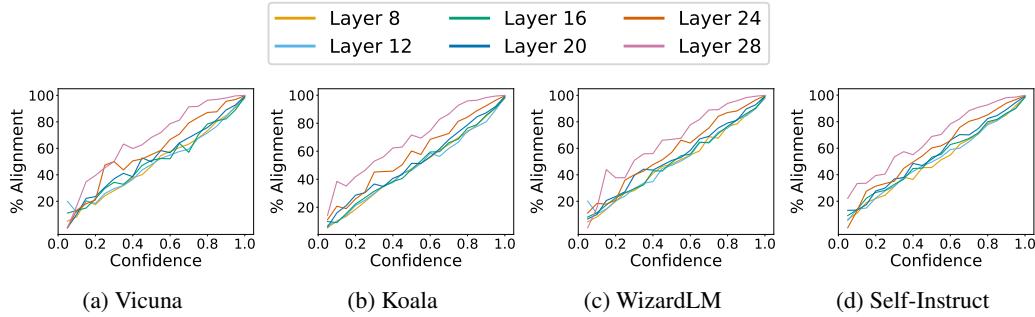


Figure 6: Demonstrating relationship between token prediction confidence of the intermediate layers and the percentage alignment with the token prediction of the final layer for model tuned with LITE. **It shows that in IT with LITE, intermediate layers' token prediction probabilities (confidences) provide a strong signal of alignment with the final layer's token prediction.**

Building on top of the two findings (in 5.3 and 5.4), we perform ‘*dynamic confidence-based early exiting*’ at token level from the intermediate layers and show that it improves the efficiency of inference while maintaining the generation quality.

## 5.5 Dynamic Confidence-Based Early Exiting Improves the Inference Efficiency While Maintaining the Generation Quality

Motivated by the findings of the previous two subsections, we perform dynamic confidence-based early exiting at token-level, i.e., we exit when the token prediction confidence of the intermediate layer is sufficiently high (thus it is likely to align with the final layer’s prediction).

To this end, from the confidence vs percentage alignment curve, we identify a confidence threshold for each layer where the alignment is  $> 95\%$ . Specifically, we use the following confidence thresholds: Layer 8: 0.95, Layer 12: 0.95, Layer 16: 0.9, Layer 20: 0.9, Layer 24: 0.8, and Layer 28: 0.7.

In the main paper, we present the results and analysis for the aforementioned configuration. However, we note that a different threshold configuration can also be used for inference. For instance, a more aggressive configuration with lower thresholds (shown in Appendix B.4) leads to even more cost improvements (49.92%); though it slightly drops the quality of generation (5.34%). The trade-off between quality and cost can be balanced depending the application requirements. For example, applications with quality tolerance or resource limitations can keep low threshold to achieve high cost improvements.

**Dynamic confidence-based early exiting:** At a selected layer, we pass its representations through the LM head, calculate the softmax logit value, and compare it with the corresponding confidence threshold. If it surpasses the threshold value then we exit from that layer and proceed to generate the next token, otherwise we repeat this process at the next selected layer.

Figure 1 (in Section 1) compares the **quality of responses** and the **inference cost** (measured in FLOPs) of the standard generation method (final layer) with our dynamic early exiting method. It shows that the **dynamic early exiting method achieves consistent and considerable cost improvements (37.86% on average) while maintaining the generation quality**. Table 2 shows the percentage improvements in inference cost for each test set individually.

We note that we use FLOPs as the metric of showcasing inference efficiency improvements because it is hardware independent unlike latency.

In the remaining of this subsection, we present a thorough analysis of the results. Specifically, we first present results at category-level for the Vicuna and WizardLM datasets (5.5.1). Then, to analyze the difference between the outputs of the two methods, we compare the semantic similarity between their responses (5.5.2). Then, we dissect the effectiveness of the dynamic early exiting method in improving the computational efficiency by showing that both the methods result in comparable number of output tokens (5.5.3). Finally, we show the percentage of exits from different layers in dynamic early exiting (5.5.4).

### 5.5.1 Quality and Inference Cost Analysis at Category Level

Vicuna and WizardLM test sets also provide the category corresponding to different test instances. To this end, we present category-level quality and inference cost results for these datasets.

**Vicuna:** Figure 7 compares the quality of responses and the inference cost of the standard generation method (final layer) with our dynamic early exiting method for different categories of Vicuna test set. On average, it results in cost improvement of 33.39%.

Test Dataset	Cost Improvement (%)
Vicuna	33.39 %
Koala	35.40 %
WizardLM	36.12 %
Self Instruct	46.54 %

Table 2: Percentage improvements in the inference cost (measured in FLOPs) with dynamic early exiting for each test dataset. On average, it results in improvement of 37.86%.

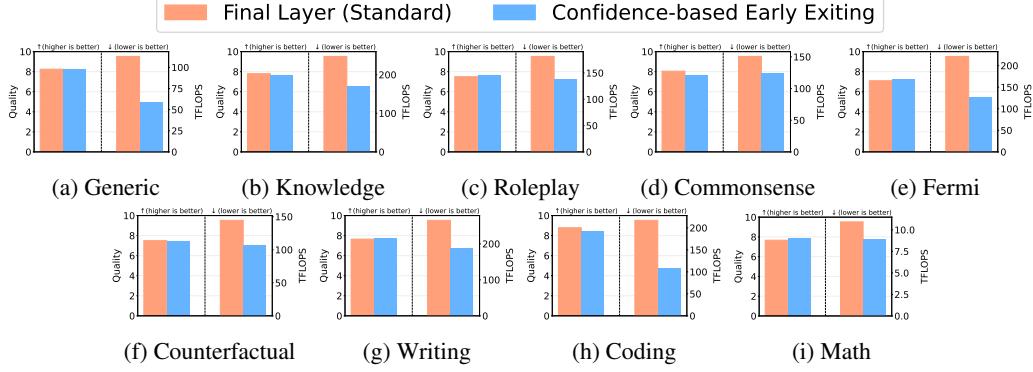


Figure 7: Comparing the quality of responses and the inference cost of the standard generation method with our dynamic early exiting method on different categories of the **Vicuna Test set**.

**WizardLM:** Figure 8 compares the quality of responses and the inference cost of the standard generation method (final layer) with our dynamic early exiting method for different categories of WizardLM test set. On average, it results in cost improvement of 36.12%.

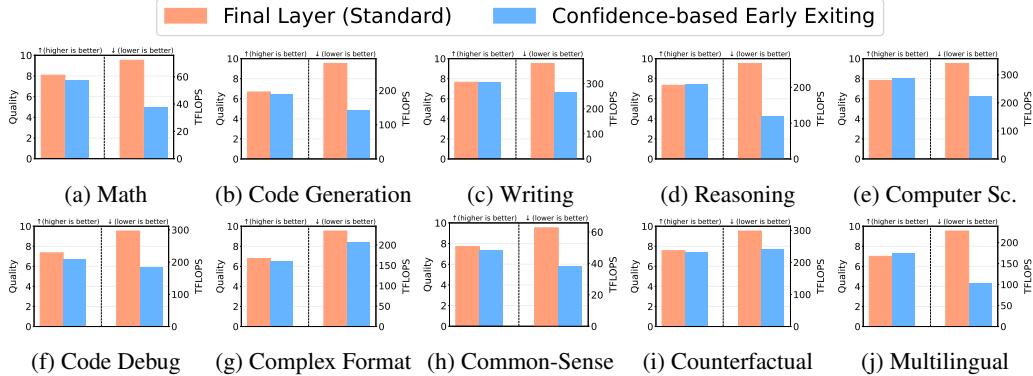


Figure 8: Comparing the quality of responses and the inference cost of the standard generation method with our dynamic early exiting method on different categories of the **WizardLM Test set**.

### 5.5.2 Dynamic Early Exiting Maintains the Semantics of the Responses

In addition to comparing the quality, we also compare the semantic similarity between the responses of the final layer and the dynamic early exiting. Table 3 shows the semantic similarity (calculated using the ‘en\_core\_web\_sm’ spacy model) for the four datasets. It shows that there is a large semantic similarity between the responses as the values are closer to 1. This implies that dynamic early exiting maintains the semantics of the responses while providing efficiency benefits.

### 5.5.3 Dissecting the Improvements in Inference Cost

In Figure 9, we compare the average number of tokens generated in the final layer’s responses and the dynamic early exiting responses. It shows that both the methods generate a comparable number of tokens in their respective outputs. **This asserts that the cost improvement resulting in**

Test Dataset	Semantic Similarity
Vicuna	0.9135
Koala	0.8940
WizardLM	0.9020
Self Instruct	0.9001

Table 3: Semantic similarity between the final layer’s responses and the dynamic early exiting responses on the four test sets. Large similarity scores (closer to 1) imply that **dynamic early exiting maintains the semantics of the responses**.

**dynamic early exiting is because of the reduced computations due to early exiting and not due to generating a lesser number of tokens.**

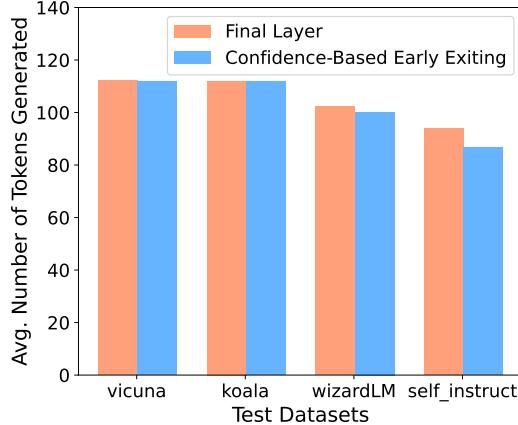


Figure 9: Comparing the average number of tokens generated in the final layer’s responses and the dynamic early exiting responses for the four datasets.

#### 5.5.4 Percentage of Token Outputs from Different Exit Layers

Figure 10 shows the percentage of token outputs from different exit layers. Note that this is aggregated across all the token positions. This shows that the model exits a considerable percentage of times from the intermediate layers (while maintaining the generation quality) which further justifies the improvement in inference efficiency.

## 6 Related Work

Improving the inference efficiency of large language models is an important research direction and is receiving considerable attention from the NLP community.

Since model size plays a major role in increasing the inference cost and latency, techniques like **quantization** [10, 45, 42, 12], **knowledge distillation** [18, 19, 22, 24], **model compression and network pruning** [39, 16, 5] have been shown to be effective in improving the inference efficiency.

Furthermore, during sampling, a cache of the keys and values can be maintained for every attention layer which reduces the computations required at inference time (**KV caching**). However, this increases the GPU VRAM memory requirement of inference.

Another technique is **speculative sampling** [20, 4] in which a short draft of  $K$  tokens is first generated from a smaller (thus faster) auto-regressive model. Then, the draft is scored using the larger model which corresponds to the target model from which we wish to sample from. Using some rejection sampling scheme, a subset of the  $K$  draft tokens is accepted by sequentially checking from left to right and thus in this process, we recover the distribution of the target model for the accepted tokens. The efficiency in this technique comes from ‘producing’ more than one token (on average) from the target model in a single pass.

**Early exiting and cascading** based inference techniques have been shown to be effective for classification tasks with BERT-style models, such as DeeBERT [43] that speeds up BERT inference by inserting extra classification layers between each encoder layer, PoWER-BERT [15] that focuses

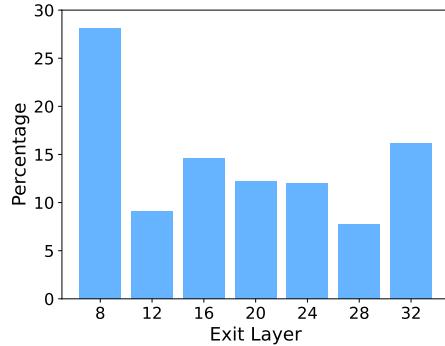


Figure 10: Percentage of token outputs from different exiting layers.

on progressive word-vector elimination (based on significance computed using self-attention) along the encoder pipeline, DynaBERT [17] that adjusts the size of the model by selecting adaptive width and depth, and cascading [33, 21, 34, 46, 6] in which sequential inference is done through models of bigger and bigger size with conditional exiting to output predictions in an efficient yet accurate manner.

## 7 Conclusion and Discussion

In this work, we proposed instruction tuning with additional explicit losses from the intermediate layers (LITE) and showed that it enables these layers to acquire ‘good’ generation ability without affecting the final layer’s generation ability. We performed ‘*dynamic confidence-based early exiting*’ at token level from the intermediate layers and showed that it improves the efficiency of inference while maintaining the generation quality. We conducted comprehensive experiments by instruction tuning LLaMA-2 models on the Alpaca dataset and evaluating on four different human-instruction test sets. We showed that dynamic early exiting achieves consistent and considerable cost improvements (37.86% on average) while maintaining the generation quality of the responses. We further conducted a thorough analysis which resulted in several important findings. Overall, our work contributes to improving the efficiency of LLM inference while maintaining the generation quality, a crucial step en route to enabling their widespread adoption.

Looking forward, our work additionally opens up several other avenues for new research, such as **speculative sampling from the intermediate layers** to improve the inference efficiency and **checking information consistency from the output of intermediate layers** to detect hallucinations.

**Speculative Sampling:** In speculative sampling [20, 4], a smaller model is used as a drafting model. However, we showed that instruction tuning with LITE enables the intermediate layers to acquire ‘good’ generation ability. Thus, an intermediate layer of the same model can be used as the draft model while the last layer remains to be the target model. This circumvents maintaining a separate drafting model for speculative sampling. In this method, KV caching can also be used.

**Hallucination Detection:** Addressing the hallucination problem of LLMs is an important research direction and a number of methods have been developed [35, 23, 1, 47, 11, 14]. One of the popular methods requires generating multiple samples and then checking the information consistency between them. Here, we can use the intermediate layers to generate the output and then check the consistency between them.

## References

- [1] Amos Azaria and Tom Mitchell. The internal state of an llm knows when its lying. *arXiv preprint arXiv:2304.13734*, 2023.
- [2] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.
- [3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.
- [4] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.
- [5] Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. The lottery ticket hypothesis for pre-trained bert networks. In H. Larochelle,

- M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 15834–15846. Curran Associates, Inc., 2020.
- [6] Zhoujun Cheng, Jungo Kasai, and Tao Yu. Batch prompting: Efficient inference with large language model apis. *arXiv preprint arXiv:2301.08721*, 2023.
  - [7] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%\* chatgpt quality, March 2023.
  - [8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.
  - [9] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.
  - [10] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.
  - [11] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. Chain-of-verification reduces hallucination in large language models. *arXiv preprint arXiv:2309.11495*, 2023.
  - [12] Elias Frantar, Saleh Ashkboos, Torsten Hoefer, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.
  - [13] Xinyang Geng, Arnav Gudibande, Hao Liu, Eric Wallace, Pieter Abbeel, Sergey Levine, and Dawn Song. Koala: A dialogue model for academic research. Blog post, April 2023.
  - [14] Zhibin Gou, Zihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*, 2023.
  - [15] Saurabh Goyal, Anamitra Roy Choudhury, Saurabh Raje, Venkatesan Chakaravarthy, Yogish Sabharwal, and Ashish Verma. Power-bert: Accelerating bert inference via progressive word-vector elimination. In *International Conference on Machine Learning*, pages 3690–3699. PMLR, 2020.
  - [16] Demi Guo, Alexander Rush, and Yoon Kim. Parameter-efficient transfer learning with diff pruning. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4884–4896, Online, August 2021. Association for Computational Linguistics.
  - [17] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. Dynabert: Dynamic bert with adaptive width and depth. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9782–9793. Curran Associates, Inc., 2020.
  - [18] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301*, 2023.
  - [19] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. TinyBERT: Distilling BERT for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4163–4174, Online, November 2020. Association for Computational Linguistics.
  - [20] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

- [21] Lei Li, Yankai Lin, Deli Chen, Shuhuai Ren, Peng Li, Jie Zhou, and Xu Sun. CascadeBERT: Accelerating inference of pre-trained language models via calibrated complete models cascade. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 475–486, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.
- [22] Zheng Li, Zijian Wang, Ming Tan, Ramesh Nallapati, Parminder Bhatia, Andrew Arnold, Bing Xiang, and Dan Roth. DQ-BART: Efficient sequence-to-sequence model via joint distillation and quantization. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 203–211, Dublin, Ireland, May 2022. Association for Computational Linguistics.
- [23] Potsawee Manakul, Adian Liusie, and Mark JF Gales. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. *arXiv preprint arXiv:2303.08896*, 2023.
- [24] Seyed Iman Mirzadeh, Mehrdad Farajtabar, Ang Li, Nir Levine, Akihiro Matsukawa, and Hassan Ghasemzadeh. Improved knowledge distillation via teacher assistant. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5191–5198, 2020.
- [25] Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. Cross-task generalization via natural language crowdsourcing instructions. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3470–3487, Dublin, Ireland, May 2022. Association for Computational Linguistics.
- [26] OpenAI. Gpt-4 technical report. *ArXiv*, abs/2303.08774, 2023.
- [27] Mihir Parmar, Swaroop Mishra, Mirali Purohit, Man Luo, Murad Mohammad, and Chitta Baral. In-BoXBART: Get instructions into biomedical multi-task learning. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 112–128, Seattle, United States, July 2022. Association for Computational Linguistics.
- [28] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.
- [29] Victor Sanh, Albert Webson, Colin Raffel, Stephen Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M Rush. Multitask prompted training enables zero-shot task generalization. In *International Conference on Learning Representations*, 2022.
- [30] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.
- [31] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. [https://github.com/tatsu-lab/stanford\\_alpaca](https://github.com/tatsu-lab/stanford_alpaca), 2023.
- [32] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.
- [33] Neeraj Varshney and Chitta Baral. Model cascading: Towards jointly improving efficiency and accuracy of NLP systems. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11007–11021, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.

- [34] Neeraj Varshney and Chitta Baral. Post-abstention: Towards reliably re-attempting the abstained instances in QA. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 967–982, Toronto, Canada, July 2023. Association for Computational Linguistics.
- [35] Neeraj Varshney, Wenlin Yao, Hongming Zhang, Jianshu Chen, and Dong Yu. A stitch in time saves nine: Detecting and mitigating hallucinations of LMs by validating low-confidence generation. *arXiv preprint arXiv:2307.03987*, 2023.
- [36] Peiyi Wang, Lei Li, Liang Chen, Dawei Zhu, Binghui Lin, Yunbo Cao, Qi Liu, Tianyu Liu, and Zhipeng Sui. Large language models are not fair evaluators. *arXiv preprint arXiv:2305.17926*, 2023.
- [37] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*, 2022.
- [38] Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Atharva Naik, Arjun Ashok, Arut Selvan Dhanasekaran, Anjana Arunkumar, David Stap, Eshaan Pathak, Giannis Karamanolakis, Haizhi Lai, Ishan Purohit, Ishani Mondal, Jacob Anderson, Kirby Kuznia, Krima Doshi, Kuntal Kumar Pal, Maitreya Patel, Mehrad Moradshahi, Mihir Parmar, Mirali Purohit, Neeraj Varshney, Phani Rohitha Kaza, Pulkit Verma, Ravsehaj Singh Puri, Rushang Karia, Savan Doshi, Shailaja Keyur Sampat, Siddhartha Mishra, Sujan Reddy A, Sumanta Patro, Tanay Dixit, and Xudong Shen. Super-NaturalInstructions: Generalization via declarative instructions on 1600+ NLP tasks. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5085–5109, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.
- [39] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. Structured pruning of large language models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6151–6162, Online, November 2020. Association for Computational Linguistics.
- [40] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*, 2022.
- [41] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.
- [42] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 38087–38099. PMLR, 23–29 Jul 2023.
- [43] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. DeeBERT: Dynamic early exiting for accelerating BERT inference. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2246–2251, Online, July 2020. Association for Computational Linguistics.
- [44] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. WizardLM: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*, 2023.
- [45] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in Neural Information Processing Systems*, 35:27168–27183, 2022.
- [46] Murong Yue, Jie Zhao, Min Zhang, Liang Du, and Ziyu Yao. Large language model cascades with mixture of thoughts representations for cost-efficient reasoning. *arXiv preprint arXiv:2310.03094*, 2023.

- [47] Tianhua Zhang, Hongyin Luo, Yung-Sung Chuang, Wei Fang, Luc Gaitskell, Thomas Hartvigsen, Xixin Wu, Danny Fox, Helen Meng, and James Glass. Interpretable unified language checking. *arXiv preprint arXiv:2304.03728*, 2023.
- [48] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *arXiv preprint arXiv:2306.05685*, 2023.

## Appendix

### A Evaluation Methodology

We use the following prompt with Claude model for comparing the responses of two models:

**Human:** You are a helpful and precise assistant for checking the quality of the answer.

**[Question]**

{question}

**[The Start of Assistant 1's Answer]**

{answer\_1}

**[The End of Assistant 1's Answer]**

**[The Start of Assistant 2's Answer]**

{answer\_2}

**[The End of Assistant 2's Answer]**

We would like to request your feedback on the performance of two AI assistants in response to the user question displayed above.

Please rate the helpfulness, relevance, accuracy, level of details of their responses. Each assistant receives an overall score on a scale of 1 to 10, where a higher score indicates better overall performance.

Please output a single line containing only two values indicating the scores for Assistant 1 and 2, respectively. The two scores are separated by a space.

**Assistant:**

We use amazon bedrock to call the Claude API with temperature set to 0 and modelId to ‘anthropic.claude-v1’.

## B Additional Results

### B.1 Inability of the Intermediate Layers of the Model Tuned with Standard Instruction Tuning to Generate ‘High-Quality’ Text

**Question:** What are the differences between plant-based and animal-based protein sources?

**Layer 32 (Final):** The main difference between plant-based and animal-based protein sources is that plant-based proteins are found in plant-based foods, such as legumes, nuts, and seeds, while animal-based proteins are found in animal-based foods, such as meat, eggs, and fish. Plant-based proteins typically contain fewer calories, fat, and cholesterol than animal-based proteins. Additionally, plant-based proteins are often fortified with essential vitamins and minerals, while animal-based proteins may not be.

**Layer 28:** plant- based proteins sources such as legumes, legumes, and nuts contain essential amino acids needed for growth and repair of cellular structures. animal based proteins sources such as meat, eggs, and fish contain additional nutrients such as vitamin B Complex vitamin D vitamin Iron and calcium needed for optimal health.

**Layer 24:** plant based prote source protein source diversity, quality proteins richness quantityquantityquantityquantity quantity quantity quantity quantity q ...

**Layer 20:** difference between Plant Based vs animal based protein source age oldenhamptonensis proportionalityuniciclesqueue ...

**Question:** Imagine you are a time traveler from the year 3000. What technological advancements would you tell people about?

**Layer 32 (Final):** From the year 3000, I would tell people about the advancements in technology that have made life easier and more efficient. I would talk about the autonomous cars, the implanted chips that allow people to access the internet and doctors to monitor patients remotely, the 3D-printed homes and the list goes on. I would also tell people about the advancements in renewable energy and how we are able to generate electricity from the sun and the wind.

**Layer 28:** From the year 3000 perspective, some technological advances might include flying cars, robotic vacuums, implanted implants used for implantable devices such as pacemakers, implanted cameras implanted microscopes implanted anything else you can think of implanted implanted implanted ...

**Layer 24:** techn adv advances techn advances techn advances techn ...

**Layer 20:** futureproofing@gmail.com domain insights intoxicatedness page entry onwardswards onwardswards onwardswards ...

## B.2 Intermediate Layers Acquire ‘Good’ Generation Ability in IT with LITE

**Question:** What are the differences between plant-based and animal-based protein sources?

**Layer 32 (Final):** Plant-based protein sources are mainly found in foods such as legumes, nuts, and seeds, while animal-based protein sources are found in foods such as meat, fish, eggs, and dairy products. Plant-based proteins are generally lower in calories and fat, and are rich in vitamins, minerals, and fiber. Animal-based proteins are typically higher in calories and fat, and are a source of essential vitamins and minerals. Plant-based proteins can also be more digestible than animal-based proteins.

**Layer 28:** Plant-based protein sources are mainly found in foods such as legumes, nuts, and seeds, while animal-based protein sources are found in foods such as meat, fish, eggs, and dairy products. Plant-based proteins are generally lower in calories and fat, and are rich in vitamins, minerals, and fiber. Animal-based proteins are typically higher in calories ...

**Layer 24:** Plant-based protein sources are mainly found in foods such as legumes, nuts, and seeds, and are typically lower in calories and fat than animal-based protein sources. Plant-based proteins are also rich in vitamins, minerals, and antioxidants, and are a good source of dietary protein. Animal-based proteins, on the other hand, are typically higher in calories and fat ...

**Layer 20:** Plant-based protein sources are found in many different foods, including legumes, nuts, seeds, and seeds. They are usually lower in calories and fat than animal-based protein sources. Plant-based proteins are also rich in fiber and contain essential vitamins and minerals that are not found in animal-based proteins. Animal-based p ...

**Question:** Imagine you are a time traveler from the year 3000. What technological advancements would you tell people about?

**Layer 32 (Final):** From the year 3000, I would tell people about the incredible advancements in technology that have been made over the centuries. We take many things for granted now, such as the internet and autonomous vehicles, but these technologies were first developed in the ...

**Layer 28:** From the year 3000, I would tell people about the advancements in artificial intelligence, robotics, and genetic engineering. In particular, I would tell people about the advancements made in AI, such as the development of autonomous cars, virtual assistants, and smart homes. I would ...

**Layer 24:** From the year 3000, I would tell people about the incredible advancements in technology that have been made over the centuries. We are now able to control our environment by using thought-controlled prostheses, use genetic ...

**Layer 20:** In the 3000s, we have developed advanced artificial intelligence, self-driving cars, and teleportation of particles. We have also made significant progress in quantum computing, which has allowed us to ...

### B.3 Relationship Between Token Prediction Confidence and Percentage Alignment of the Intermediate Layers for the Model Tuned with Instruction Tuning (IT)

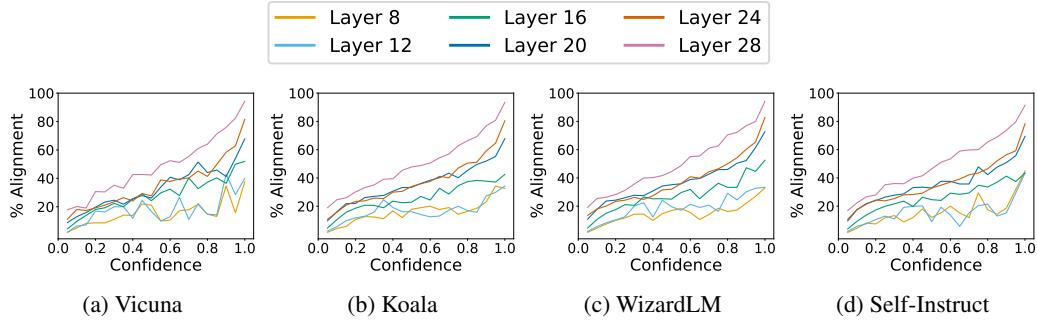


Figure 11: Demonstrating relationship between token prediction confidence of the intermediate layers and the percentage alignment with the token prediction of the final layer for model tuned with IT.

Figure 11 shows the relationship between the token prediction confidence of the intermediate layers and the percentage alignment with the token prediction of the final layer for standard instruction tuning (IT). It shows that the confidence is not well correlated with the percentage alignment. However, in IT with LITE (Figure 6), the intermediate layers' token prediction probabilities provide a strong signal of alignment.

## B.4 Dynamic Confidence-Based Early Exiting with Aggressive Confidence Thresholds

We also experiment with aggressive confidence thresholds. Specifically, we use the following confidence thresholds: Layer 8: 0.85, Layer 12: 0.85, Layer 16: 0.8, Layer 20: 0.8, Layer 24: 0.7, and Layer 28: 0.6. These thresholds are lower than those used in the main paper. Figure 12 shows the quality and cost comparisons. It leads to larger cost improvements (of 49.92%) though it slightly drops the quality of generation (by 5.34%).

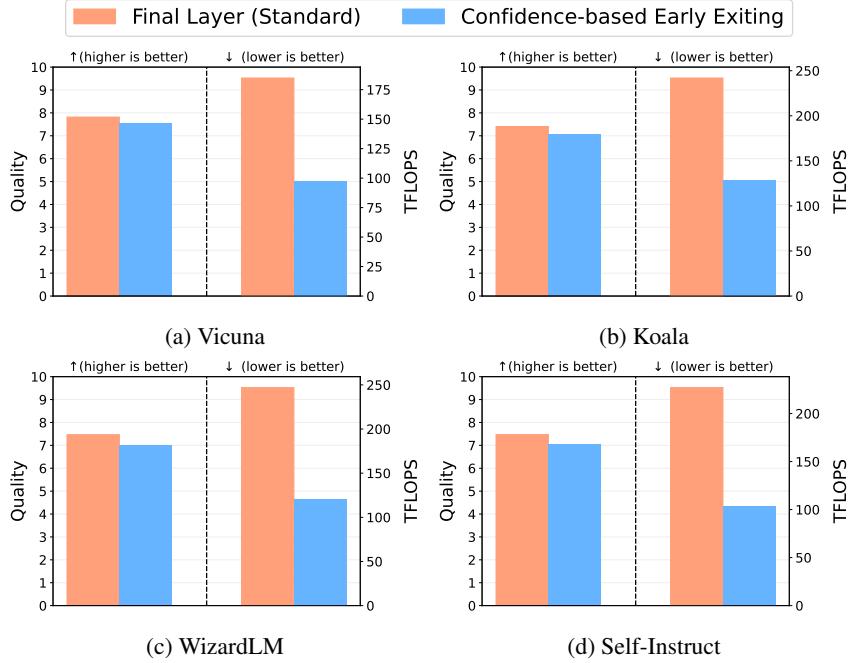


Figure 12: Comparing the quality of responses (evaluated using the Claude model) and the inference cost (measured in FLOPs) of the standard generation method from the final layer with our dynamic early exiting method. Confidence Thresholds: Layer 8: 0.85, Layer 12: 0.85, Layer 16: 0.8, Layer 20: 0.8, Layer 24: 0.7, and Layer 28: 0.6. This aggressive configuration results in larger cost improvements of 49.93% but results in a slight degradation in the generation quality.

# AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration

Ji Lin<sup>1\*</sup> Jiaming Tang<sup>1,2\*</sup> Haotian Tang<sup>1</sup> Shang Yang<sup>1</sup> Xingyu Dang<sup>3</sup> Chuang Gan<sup>1</sup> Song Han<sup>1</sup>

<sup>1</sup>MIT <sup>2</sup>SJTU <sup>3</sup>Tsinghua University

<https://github.com/mit-han-lab/llm-awq>

## Abstract

Large language models (LLMs) have shown excellent performance on various tasks, but the astronomical model size raises the hardware barrier for serving (memory size) and slows down token generation (memory bandwidth). In this paper, we propose Activation-aware Weight Quantization (AWQ), a hardware-friendly approach for LLM low-bit weight-only quantization. Our method is based on the observation that weights are not equally important: *protecting only 1% of salient weights can greatly reduce quantization error*. We then propose to search for the optimal per-channel scaling that protects the salient weights by observing the *activation, not weights*. AWQ does not rely on any backpropagation or reconstruction, so it can well preserve LLMs' generalization ability on different domains and modalities, without overfitting to the calibration set. AWQ outperforms existing work on various language modeling and domain-specific benchmarks. Thanks to better generalization, it achieves excellent quantization performance for *instruction-tuned* LMs and, for the first time, *multi-modal* LMs. Alongside AWQ, we implement an efficient and flexible inference framework tailored for LLMs on the edge, offering more than  $3\times$  speedup over the Huggingface FP16 implementation on both desktop and mobile GPUs. It also democratizes the deployment of the 70B Llama-2 model on mobile GPU (NVIDIA Jetson Orin 64GB).

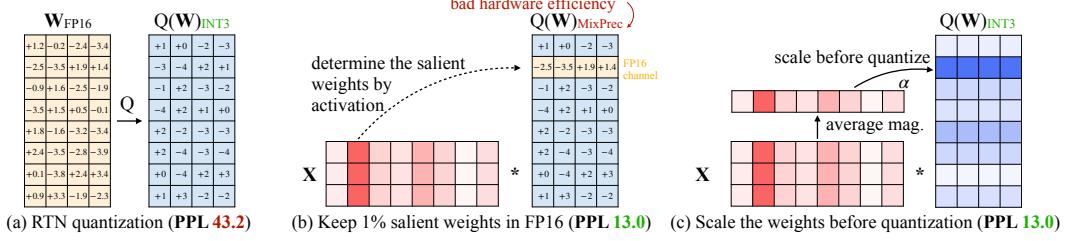
## 1 Introduction

Large language models (LLMs) based on transformers [40] have shown excellent performance on various benchmarks [4, 49, 38, 34]. However, the large model size leads to the high serving costs. For example, GPT-3 has 175B parameters, which is 350GB in FP16, while the latest H100 GPU only has 96GB memory, let alone edge devices.

Low-bit weight quantization for LLMs can save memory but is hard. Quantization-aware training (QAT) is not practical due to the high training cost, while post-training quantization (PTQ) suffers from large accuracy degradation under a low-bit setting. The closest work is GPTQ [14], which uses second-order information to perform error compensation. It may over-fit the calibration set during reconstruction, distorting the learned features on out-of-distribution domains (Figure 6), which could be problematic since LLMs are *generalist* models.

In this paper, we propose Activation-aware Weight Quantization (AWQ), a hardware-friendly low-bit weight-only quantization method for LLMs. Our method is based on the observation that *weights are not equally important* for LLMs' performance. There is a small fraction (0.1%-1%) of *salient* weights; skipping the quantization of these salient weights will significantly reduce the quantization loss (Table 1). To find the salient weight channels, the insight is that we should refer to the *activation* distribution instead of the *weight* distribution, despite we are doing *weight-only* quantization: weight channels corresponding to larger activation magnitudes are more salient since they process more

\* indicates equal contributions.



**Figure 1.** We observe that we can find 1% of the salient weights in LLMs by observing the *activation distribution* (middle). Keeping the salient weights in FP16 can significantly improve the quantized performance (PPL from 43.2 (left) to 13.0 (middle)), but the mixed-precision format is not hardware-efficient. We follow the activation-awareness principle and propose AWQ (right). AWQ performs per-channel scaling to protect the salient weights, leading to reduced quantized error. PPL is measured with OPT-6.7B under INT3-g128 quantization.

important features. To avoid the hardware-inefficient mixed-precision implementation, we analyze the error from weight quantization and derive that *scaling up the salient channels can reduce their relative quantization error* (Equation 2). Following the intuition, we designed a per-channel scaling method to automatically search for the optimal scaling that minimizes the quantization error under full-weight quantization. AWQ does not rely on any backpropagation or reconstruction, so it can well preserve LLMs’ generalization ability on various domains and modalities without overfitting to the calibration set. Furthermore, we implemented an efficient serving framework to convert theoretical memory savings from AWQ to practical speedup. Our framework takes advantage of kernel fusion to minimize the inference overhead (*e.g.*, intermediate DRAM access and kernel launch overhead), so that we can better realize the speed up from quantizing linear layers (AWQ is applied to linear layers which consist most of the parameters).

Experiments show that AWQ outperforms existing work on various tasks for different model families (*e.g.*, LLaMA [38], OPT [49]) and model sizes. Thanks to better generalization, it also achieves good quantization performance for *instruction-tuned* LMs (*e.g.*, Vicuna) and, for the first time, *multi-modal* LMs (OpenFlamingo [2]). With our efficient system implementation, we consistently observe a **3.2-3.3 $\times$**  average speedup compared to the FP16 implementation by Huggingface across a diverse spectrum of LLMs. Furthermore, it facilitates effortless deployment of the Llama-2-70B model on a single NVIDIA Jetson Orin with 64GB of memory. It also democratizes LLMs with up to 13 billion parameters at an interactive pace of 30 tokens per second on a laptop RTX 4070 GPU with only 8GB of memory.

AWQ has been widely adopted by various open-source LLM serving solutions including [FastChat](#), [vLLM](#), [HuggingFace TGI](#), [LMDeploy](#), etc.

## 2 AWQ: Activation-aware Weight Quantization

*Quantization* maps a floating-point number into lower-bit integers. It is an effective method to reduce the model size and inference costs of LLMs [9, 14, 47, 46]. In this section, we first propose a weight-only quantization method to improve accuracy *without training/regression* by protecting more “important” weights. And then develop a data-driven method to search for the optimal scaling that reduces quantization errors (Figure 1).

### 2.1 Improving LLM Quantization by Preserving 1% Salient Weights

We observe that the weights of LLMs are *not equally important*: there is a small fraction of *salient* weights that are much more important for LLMs’ performance compared to others. Skipping the quantization of these salient weights can help bridge the performance degradation due to the quantization loss *without* any training or regression (Figure 1(b)). To verify the idea, we benchmark the performance of quantized LLMs when skipping part of the weight channels in Table 1. We measured the performance of INT3 quantized models while keeping some ratios of weight channels in FP16. A widely used method to determine the importance of weights is to look at its magnitude or  $L_2$ -norm [18, 13]. But we find skipping the weight channels with large norm (*i.e.*, FP16% (based on W)) does not significantly improve the quantized performance, leading to a similar marginal improvement as random selection. Interestingly, selecting weights based on *activation magnitude* can significantly improve the performance: keeping only 0.1%-1% of the channels corresponding to larger

PPL ↓	FP16	RTN (w3-g128)	FP16% (based on act.)			FP16% (based on W)			FP16% (random)		
			0.1%	1%	3%	0.1%	1%	3%	0.1%	1%	3%
OPT-1.3B	14.62	119.00	25.03	16.91	16.68	108.71	98.55	98.08	119.76	109.38	61.49
OPT-6.7B	10.86	23.54	11.58	11.39	11.36	23.41	22.37	22.45	23.54	24.23	24.22
OPT-13B	10.13	46.04	10.51	10.43	10.42	46.07	48.96	54.49	44.87	42.00	39.71

**Table 1.** Keeping a small fraction of weights (0.1%-1%) in FP16 significantly improves the performance of the quantized models over round-to-nearest (RTN). It is only effective when we select the important weights in FP16 by looking at *activation* distribution instead of *weight* distribution. We highlight results with a decent perplexity in green. We used INT3 quantization with a group size of 128 and measured the WikiText perplexity (↓).

activation significantly improves the quantized performance, even matching a strong reconstruction-based method GPTQ [14]. We hypothesize that the input features with larger magnitudes are generally more important. Keeping the corresponding weights in FP16 can preserve those features, which contributes to better model performance.

**Limitations:** Despite keeping 0.1% of weights in FP16 can improve the quantized performance without a noticeable increase in model size (measured in total bits), such a mixed-precision data type will make the system implementation difficult. We need to come up with a method to protect the important weights without actually keeping them as FP16.

## 2.2 Protecting Salient Weights by Activation-aware Scaling

We propose an alternative method to reduce the quantization error of the salient weight by *per-channel scaling*, which does not suffer from the hardware inefficiency issue.

**Analyzing the quantization error.** We start by analyzing the error from weight-only quantization. Consider a group/block of weight  $\mathbf{w}$ ; the linear operation can be written as  $y = \mathbf{w}\mathbf{x}$ , and the quantized counterpart is  $y = Q(\mathbf{w})\mathbf{x}$ . Specifically, the quantization function is defined as:

$$Q(\mathbf{w}) = \Delta \cdot \text{Round}\left(\frac{\mathbf{w}}{\Delta}\right), \quad \Delta = \frac{\max(|\mathbf{w}|)}{2^{N-1}}, \quad (1)$$

where  $N$  is the number of quantization bits, and  $\Delta$  is the quantization scaler determined by the absolute maximum value. Now consider a weight element  $w \in \mathbf{w}$ , if we multiply  $w$  with  $s > 1$  and the inversely scale  $x$ , we will have  $Q(w \cdot s)(x/s)$ , which is:

$$Q(w \cdot s) \cdot \frac{x}{s} = \Delta' \cdot \text{Round}\left(\frac{ws}{\Delta}\right) \cdot x \cdot \frac{1}{s}, \quad (2)$$

where  $\Delta'$  is the new quantization scaler after applying  $s$ . We empirically find that: (1) The expected error from  $\text{Round}(\cdot)$  (denoted as *RoundErr*) does not vary: since the round function maps a floating-point number to an integer, the error is roughly uniformly distributed from 0-0.5, resulting in an average error of 0.25; (2) Scaling up a single element  $w$  usually does not change the extreme value from the group  $\mathbf{w}$ . Therefore we have  $\Delta' \approx \Delta$ ; (3) The error from equation 2 can be expressed as  $\text{Err}' = \Delta' \cdot \text{RoundErr} \cdot \frac{1}{s}$ , the ratio compared to the original error *RoundErr* is  $\frac{\Delta'}{\Delta} \cdot \frac{1}{s}$ . Given  $\Delta' \approx \Delta$  and  $s > 1$ , the relative error is smaller for the salient weight  $w$ .

To verify the idea, we multiply the 1% salient channels with  $s > 1$  for the OPT-6.7B model, and measure the change in  $\Delta$  for each group in Table 2. We find that scaling up the salient channels is quite effective: the perplexity improves from 23.54 for  $s = 1$  (simply RTN) to 11.92 for  $s = 2$ . As  $s$  goes larger, the percentage of changed  $\Delta$  generally gets larger, but the proportion is still quite small for  $s < 2$ ; the relative error for the salient channels continues to go smaller as  $s$  increases. Nonetheless, the best PPL actually appears at  $s = 2$ . This is because if we use a very large  $s$ , it will increase the relative error for the *non-salient* channels when  $\Delta$  increases (the error of non-salient channels will be amplified by  $\frac{\Delta'}{\Delta}$ , and the ratio is larger than 1 for 21.2% of the channels under  $s = 4$ ), which can damage the model's overall accuracy. Therefore, we need to also consider the error from the non-salient channels when protecting salient ones.

**Searching to scale.** To consider both salient and non-salient weights, we choose to automatically search for an optimal (per input channel) scaling factor that minimizes the output difference after

<b>OPT-6.7B</b>	$s = 1$	$s = 1.25$	$s = 1.5$	$s = 2$	$s = 4$
proportion of $\Delta' \neq \Delta$	0%	2.8%	4.4%	8.2%	21.2%
average $\Delta'/\Delta$	1	1.005	1.013	1.038	1.213
average $\frac{\Delta'}{\Delta} \cdot \frac{1}{s}$ (error reduction rate)	1	0.804	0.676	0.519	<b>0.303</b>
Wiki-2 PPL	23.54	12.87	12.48	<b>11.92</b>	12.36

**Table 2.** Statistics when multiplying the 1% salient channels by  $s > 1$ . Scaling up the salient channels significantly improves the perplexity (23.54 to 11.92). As  $s$  goes larger, the percentage of changed  $\Delta$  increases, and the error reduction rate for salient channels also increases. However, the best perplexity is achieved at  $s = 2$ , since further increasing  $s$  will increase the quantization error for *non-salient* channels.

<b>OPT / PPL↓</b>	1.3B	2.7B	6.7B	13B	30B
FP16	-	14.62	12.47	10.86	10.13
	RTN	119.47	298.00	23.54	46.04
INT3	1% FP16	16.91	13.69	<b>11.39</b>	<b>10.43</b>
g128	$s = 2$	18.63	14.94	11.92	10.80
	AWQ	<b>16.32</b>	<b>13.58</b>	<b>11.39</b>	10.56
					<b>9.77</b>

**Table 3.** AWQ protects salient weights and reduces quantization error by using a scaling-based method. It consistently outperforms Round-to-nearest quantization (RTN) and achieves comparable performance as mixed-precision (1% FP16) while being more hardware-friendly.

quantization for a certain layer. Formally, we want to optimize the following objective:

$$\mathbf{s}^* = \arg \min_{\mathbf{s}} \mathcal{L}(\mathbf{s}), \quad \mathcal{L}(\mathbf{s}) = \|Q(\mathbf{W} \cdot \mathbf{s})(\mathbf{s}^{-1} \cdot \mathbf{X}) - \mathbf{WX}\| \quad (3)$$

Here  $Q$  means the weight quantization function (*e.g.*, INT3/INT4 quantization with group size 128),  $\mathbf{W}$  is the original weights in FP16, and  $\mathbf{X}$  is the input features cached from a small calibration set (we take a small calibration set from the pre-training dataset in order not to overfit to a specific task).  $\mathbf{s}$  is a per-(input) channel scaling factor; for  $\mathbf{s}^{-1} \cdot \mathbf{X}$ , it can usually be fused into the previous operator [44, 46]. Since the quantization function is not differentiable, we are not able to directly optimize the problem with vanilla backpropagation. There are some techniques relying on approximated gradients [3, 12], which we found still suffers from unstable convergence.

To make the process more stable, we define a *search space* for the optimal scale by analyzing the factors that will affect the choice of scaling factor. As shown in the last section, the saliency of weight channels is actually determined by the activation scale (thus “activation-awareness”). Therefore, we simply use a very simple search space:

$$\mathbf{s} = \mathbf{s}_{\mathbf{X}}^{\alpha}, \quad \alpha^* = \arg \min_{\alpha} \mathcal{L}(\mathbf{s}_{\mathbf{X}}^{\alpha}) \quad (4)$$

$\mathbf{s}$  is only related to the magnitude of activation  $\mathbf{s}_{\mathbf{X}}$ , and we use a single hyper-parameter  $\alpha$  to balance between the protection of salient and non-salient channels. We can find the best  $\alpha$  by a fast grid search over the interval of  $[0, 1]$  (0 means we do not scale; 1 corresponds to the most aggressive scaling). We further apply weight clipping also by minimizing the MSE error, since clipping the weights can further help to reduce  $\Delta'$  in Equation 2; thus reducing quantization error. We provide an ablation study on OPT models under INT3-g128 quantization in Table 3; AWQ consistently outperforms round-to-nearest quantization (RTN) and achieves comparable performance as mixed-precision (1% FP16) while being more hardware-friendly.

**Advantages.** Our method does not rely on any regression [14] or backpropagation, which is required by many quantization-aware training methods. It has minimal reliance on the calibration set since we only measure the average magnitude per channel, thus preventing over-fitting (Figure 6). Therefore, our method requires fewer data for the quantization process and can preserve LLMs’ knowledge outside of the calibration set’s distribution. See Section 3.3 for more details.

PPL $\downarrow$		Llama-2			LLaMA			
		7B	13B	70B	7B	13B	30B	65B
FP16	-	5.47	4.88	3.32	5.68	5.09	4.10	3.53
INT3 g128	RTN	6.66	5.52	3.98	7.01	5.88	4.88	4.24
	GPTQ	6.43	5.48	3.88	8.81	5.66	4.88	4.17
	GPTQ-R	6.42	5.41	3.86	6.53	5.64	4.74	4.21
	AWQ	<b>6.24</b>	<b>5.32</b>	<b>3.74</b>	<b>6.35</b>	<b>5.52</b>	<b>4.61</b>	<b>3.95</b>
INT4 g128	RTN	5.73	4.98	3.46	5.96	5.25	4.23	3.67
	GPTQ	5.69	4.98	3.42	6.22	5.23	4.24	3.66
	GPTQ-R	5.63	4.99	3.43	5.83	5.20	4.22	3.66
	AWQ	<b>5.60</b>	<b>4.97</b>	<b>3.41</b>	<b>5.78</b>	<b>5.19</b>	<b>4.21</b>	<b>3.62</b>

**Table 4.** AWQ improves over round-to-nearest quantization (RTN) for different model sizes and different bit-precisions. It consistently achieves better perplexity than GPTQ (w/ and w/o reordering) on LLaMA & Llama-2 models.

### 3 Experiments

#### 3.1 Settings

**Quantization.** We focus on *weight-only grouped* quantization in this work. As shown in previous work [10, 14], grouped quantization is always helpful for improving performance/model size trade-off. We used a group size of 128 throughout the work, except otherwise specified. We focus on INT4/INT3 quantization since they are able to mostly preserve the LLMs’ performance [10]. For AWQ, we used a small calibration set from the Pile [15] dataset in order not to overfit to a specific downstream domain. We used a grid size of 20 to search for the optimal  $\alpha$  in Equation 4.

**Models.** We benchmarked our method on LLaMA [38] and OPT [49] families. There are other open LLMs like BLOOM [34], but they are generally worse in quality, so we do not include them in our study. We further benchmark an instruction-tuned model Vicuna [6] and visual language models OpenFlamingo-9B [2] and LLaVA-13B [26] to demonstrate the generability of our method.

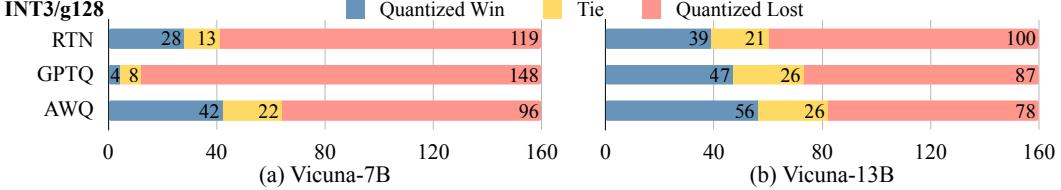
**Evaluations.** Following previous literature [9, 46, 14, 10, 47], we mainly profiled the quantized models on language modeling tasks (perplexity evaluation on WikiText-2 [27]) since perplexity can stably reflect the LLM’s performance [10].

**Baselines.** Our primary baseline is vanilla round-to-nearest quantization (RTN). It is actually quite strong when using a small group size like 128 [14, 10]. We also compare with a state-of-the-art method GPTQ [14] for LLM weight quantization. For GPTQ, we also compare with an updated version that uses a “reorder” trick (denoted as GPTQ-Reorder or GPTQ-R). Other techniques like ZeroQuant [47], AdaRound [28], and BRECQ [23] rely on backpropagation to update the quantized weights, which may not easily scale up to large model sizes; they also do not outperform GPTQ [14], thus not included for study.

#### 3.2 Evaluation

**Results on LLaMA models.** We focus our study on LLaMA models (LLaMA [38] and Llama-2 [39]) due to their superior performance compared to other open-source LLMs [49, 34]; it is also the foundation of many popular open-source models [36, 6]. We evaluate the perplexity before and after quantization in Table 4. We can see that AWQ consistently outperforms round-to-nearest (RTN) and GPTQ [14] (w/ and w/o reordering) across different model scales (7B-70B) and generations.

**Quantization of instruction-tuned models.** Instruction tuning can significantly improve the models’ performance and usability [42, 33, 31, 8]. It has become an essential procedure before model deployment. We further benchmark our method’s performance on a popular instruction-tuned model Vicuna [6] in Figure 2. We used the GPT-4 score to evaluate the quantized models’ performance against the FP16 counterpart on 80 sample questions [6]. We compare the responses with both orders (quantized-FP16, FP16-quantized) to get rid of the ordering effect (we found GPT-4 tends to increase the rating of the first input), leading to 160 trials. AWQ consistently improves



**Figure 2.** Comparing INT3-g128 quantized Vicuna models with FP16 counterparts under GPT-4 evaluation protocol [6]. More winning cases (in blue) indicate better performance. AWQ consistently improves the quantized performance compared to RTN and GPTQ [14], showing generalization to instruction-tuned models.

COCO (CIDEr $\uparrow$ )		0-shot	4-shot	8-shot	16-shot	32-shot	$\Delta(32\text{-shot})$
FP16	-	63.73	72.18	76.95	79.74	81.70	-
INT4 g128	RTN	60.24	68.07	72.46	74.09	77.13	-4.57
	GPTQ	59.72	67.68	72.53	74.98	74.98	-6.72
	AWQ	<b>62.57</b>	<b>71.02</b>	<b>74.75</b>	<b>78.23</b>	<b>80.53</b>	<b>-1.17</b>
INT3 g128	RTN	46.07	55.13	60.46	63.21	64.79	-16.91
	GPTQ	29.84	50.77	56.55	60.54	64.77	-16.93
	AWQ	<b>56.33</b>	<b>64.73</b>	<b>68.79</b>	<b>72.86</b>	<b>74.47</b>	<b>-7.23</b>

**Table 5.** Quantization results of a visual language model OpenFlamingo-9B [2] on COCO Captioning datasets. AWQ outperforms existing methods under zero-shot and various few-shot settings, demonstrating the generability to different modalities and in-context learning workloads. AWQ reduces the quantization degradation (32-shot) from 4.57 to 1.17 under INT4-g128, providing 4 $\times$  model size reduction with negligible performance loss.

the INT3-g128 quantized Vicuna models over RTN and GPTQ under both scales (7B and 13B), demonstrating the generability to instruction-tuned models.

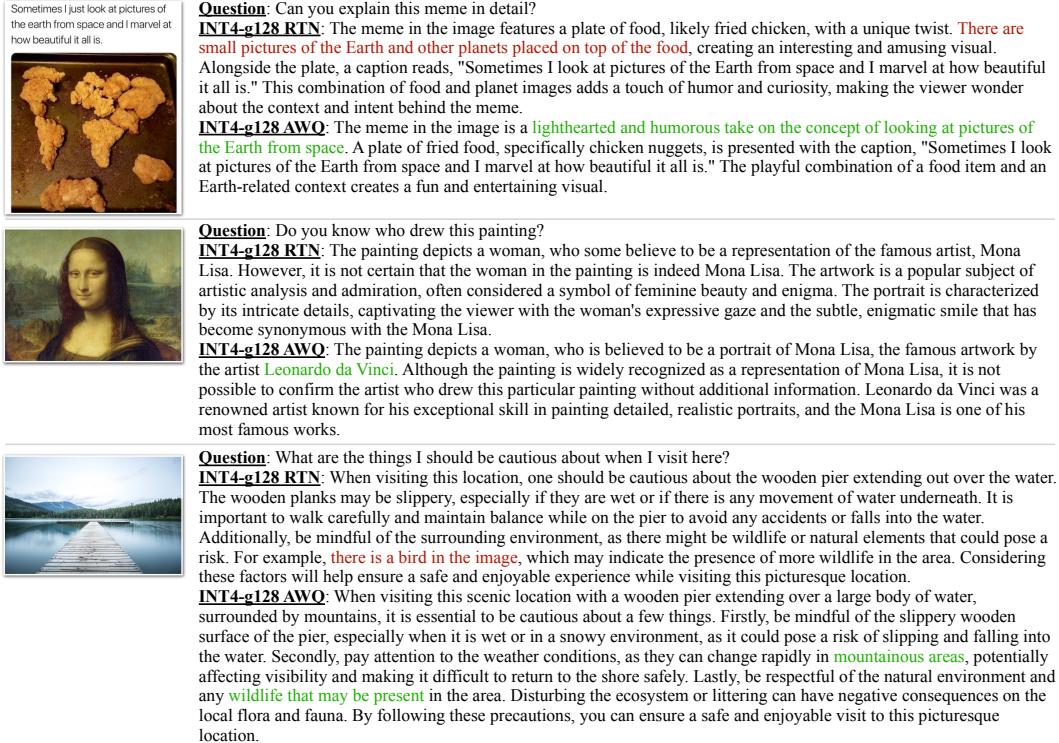
**Quantization of multi-modal language models.** Large multi-modal models (LMMs) or visual language models (VLMs) are LLMs augmented with vision inputs [1, 22, 21, 11, 48, 26]. Such models are able to perform text generation conditioned on image/video inputs. Since our method does not have the overfitting issue to the calibration set, it can be directly applied to VLMs to provide accurate and efficient quantization. We perform experiments with the OpenFlamingo-9B model [2] (an open-source reproduction of [1]) on COCO captioning [5] dataset (Table 5). We measured the average performance of 5k samples under different few-shot settings. We only quantize the language part of the model since it dominates the model size. AWQ outperforms existing methods under zero-shot and various few-shot settings, demonstrating the generability to different modalities and in-context learning workloads. It reduces the quantization degradation (32-shot) from 4.57 to 1.17 under INT4-g128, providing 4 $\times$  model size reduction with negligible performance loss. We further provide some qualitative captioning results in Figure 3 to show our advantage over RTN. Our method provides a push-the-button solution for LMM/VLM quantization. It is the *first* study of VLM low-bit quantization to the best of our knowledge.

**Visual reasoning results.** We further provide some qualitative visual reasoning examples of the LLaVA-13B [26] model in Figure 4. AWQ improves the responses compared to the round-to-nearest (RTN) baseline for INT4-g128 quantization, leading to more reasonable answers. In this first example, the AWQ model can understand the meme as it resembles the Earth when looking from space, while RTN produces wrong descriptions (marked in red). In the second example, AWQ correctly answers the question (the artist of the painting), while RTN does not provide any information about the artist. In the last example, RTN falsely points out a bird in the picture, while AWQ provides more information by noticing the image is taken in a mountain area. AWQ improves the visual reasoning ability of VLMs by reducing factual errors in the responses; RTN is not good enough even for 4 bits.

**Extreme low-bit quantization.** We further quantize LLM to INT2 to accommodate limited device memory (Table 6). RTN completely fails, and AWQ brings significant perplexity improvement on top of GPTQ, though there is still a performance gap compared to FP16. Our method is orthogonal to GPTQ. We can combine our method with GPTQ to further improve the INT2 quantization performance, making it a more practical setting.



**Figure 3.** Qualitative results of quantized OpenFlamingo-9B [2] on COCO captioning dataset (4-shot, INT4-g128 quantization). Our method significantly improves the captioning quality compared to the round-to-nearest (RTN) baseline. We color the text to show the **correct** or **wrong** captions.



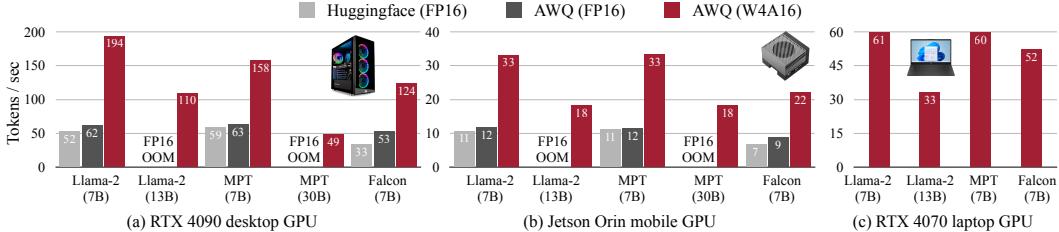
**Figure 4.** Visual reasoning examples from LLaVA-13B model [26]. AWQ improves over the round-to-nearest (RTN) baseline, providing more reasonable answers. We color the text to show the **correct** or **wrong** responses.

**Speedup Evaluation.** In Figure 5, we demonstrate the system acceleration results for AWQ. We optimize both linear layers and layers that do not have quantized weights. We conduct benchmarking experiments on RTX 4090 (desktop GPU), RTX 4070 (laptop GPU) and Jetson Orin (mobile GPU). We perform batch size = 1 inference for all LLMs using a fixed prompt length of 4 tokens. We generate 200 tokens for each inference run and calculate the median latency as the final result. As in Figure 5(a), our system brings **2.7-3.9 $\times$**  speedup to three families of LLMs (Llama-2, MPT and Falcon) on 4090 compared with the Huggingface FP16 implementation. Notably, on the laptop 4070 GPU with only 8GB memory, we are still able to run Llama-2-13B models at 33 tokens / second, while the FP16 implementation cannot fit 7B models.

Our system also exhibits promising performance on the NVIDIA Jetson Orin (32GB). As shown in Figure 5(b), our system achieves an interactive processing rate of **33 tokens per second** when running Llama-2 models. Thanks to AWQ, even larger models such as MPT-30B can operate smoothly on this resource-constrained edge device, delivering a processing speed of 7.8 tokens per second. It's worth noting that we implement the forward pass for all AWQ models using native PyTorch APIs, and this code is reused across various GPU architectures. Consequently, our system provides the best of both worlds: state-of-the-art inference speed and exceptional extensibility.

OPT / Wiki PPL $\downarrow$		1.3B	2.7B	6.7B	13B	30B
FP16	-	14.62	12.47	10.86	10.13	9.56
INT2 g64	RTN	10476	193210	7622	17564	8170
	GPTQ	46.67	28.15	16.65	16.74	11.75
AWQ +GPTQ		<b>35.71</b>	<b>25.70</b>	<b>15.71</b>	<b>13.25</b>	<b>11.38</b>

**Table 6.** Our method is orthogonal to GPTQ: it further closes the performance gap under extreme low-bit quantization (INT2-g64) when combined with GPTQ. Results are WikiText-2 perplexity of OPT models.



**Figure 5.** AWQ provides a turn-key solution to transform the theoretical memory footprint reduction into a quantifiable speedup. As a result, AWQ is up to **3.9×** and **3.5×** faster than the FP16 implementation from Huggingface on 4090 (desktop GPU) and Orin (mobile GPU), respectively. AWQ also democratizes Llama-2 13B deployment on laptop GPUs (4070) with merely 8GB memory.

### 3.3 Analysis

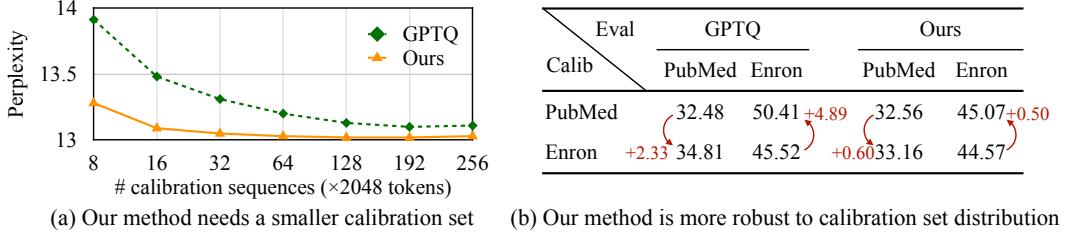
**Better data-efficiency for the calibration set.** Our method requires a smaller calibration set since we do not rely on regression/backpropagation; we only measure the average activation scale from the calibration set, which is data-efficient. To demonstrate the idea, we compare the perplexity of the OPT-6.7B model with INT3-g128 quantization in Figure 6 (a). AWQ needs a much smaller calibration to reach a good quantized performance; it can achieve better perplexity using 10× smaller calibration set compared to GPTQ (16 sequences v.s. 192 sequences).

**Robust to the calibration set distributions.** Our method is less sensitive to the calibration set distribution since we only measure the average activation scale from the calibration set, which is more generalizable across different dataset distributions. We further benchmarked the effect of the different calibration set distributions in Figure 6(b). We took two subsets from the Pile dataset [15]: PubMed Abstracts and Enron Emails [20]. We use each of the subsets as the calibration set and evaluate the quantized model on both sets (the calibration and evaluation sets are split with no overlapping; we used 1k samples for evaluation). Overall, using the same calibration and evaluation distribution works the best (PubMed-PubMed, Enron-Enron). But when using a different calibration distribution (PubMed-Enron, Enron-PubMed), AWQ only increases the perplexity by 0.5-0.6, while GPTQ has 2.3-4.9 worse perplexity. This demonstrates the robustness of AWQ to the calibration set distribution.

## 4 Related Work

**Model quantization methods.** Quantization reduces the bit-precision of deep learning models [17, 19, 29, 41, 28, 25], which helps to reduce the model size and accelerate inference. Quantization techniques generally fall into two categories: quantization-aware training (QAT, which relies on backpropagation to update the quantized weights) [3, 16, 30, 7] and post-training quantization [19, 29, 28] (PTQ, usually training-free). The QAT methods cannot easily scale up to large models like LLMs. Therefore, people usually use PTQ methods to quantize LLMs.

**Quantization of LLMs.** People study two settings for LLM quantization: (1) W8A8 quantization, where both activation and weights are quantized to INT8 [9, 46, 47, 45, 43]; (2) Low-bit weight-only quantization (*e.g.*, W4A16), where only weights are quantized into low-bit integers [14, 10, 35, 32]. We focus on the second setting in this work since it not only reduces the hardware barrier (requiring a smaller memory size) but also speeds up the token generation (remedies memory-bound workload). Apart from the vanilla round-to-nearest baseline (RTN), GPTQ [14] is the closest to our work. However, the reconstruction process of GPTQ leads to an over-fitting issue to the calibration set and



**Figure 6.** **Left:** AWQ needs a much smaller calibration set to reach a good quantized performance. It can achieve better perplexity using 10× smaller calibration set compared to GPTQ. **Right:** Our method is more robust to the calibration set distribution. Overall, using the same calibration and evaluation distribution works the best (PubMed-PubMed, Enron-Enron). But when using a different calibration and evaluation distribution (PubMed-Enron, Enron-PubMed), AWQ only increases the perplexity by 0.5-0.6, while GPTQ has 2.3-4.9 worse perplexity. All experiments are done with the OPT-6.7B model under INT3-g128 quantization.

may not preserve the generalist abilities of LLMs for other modalities and domains. It also requires a reordering trick to work for some models (*e.g.*, LLaMA-7B [38] and OPT-66B [49]).

**System support for low-bit quantized LLMs.** Low-bit quantized LLMs have been a popular setting to reduce inference costs. There are some system supports to achieve a practical speed-up. GPTQ [14] provides INT3 kernels for OPT models and GPTQ-for-LLAMA extends kernel support for INT4 reordered quantization with the help of Triton [37]. FlexGen [35] and `llama.cpp`<sup>\*</sup> perform group-wise INT4 quantization to reduce I/O costs and offloading. FasterTransformer<sup>†</sup> implements FP16×INT4 GEMM for weight-only per-tensor quantization but does not support group quantization. LUT-GEMM [32] performs bitwise computation on GPU CUDA cores with the help of lookup tables. AWQ kernels are adaptively executed on both tensor cores and CUDA cores, suitable for both context and generation phases in LLM inference. Consequently, we run state-of-the-art LLaMA models with **3.2-3.3×** speedup over the FP16 implementation from Huggingface.

## 5 Conclusion

In this work, we propose Activation-aware Weight Quantization (AWQ), a simple yet effective method for low-bit weight-only LLM compression. AWQ is based on the observation that weights are not equally important in LLMs and performs per-channel scaling to reduce the quantization loss of salient weights. AWQ does not over-fit the calibration set and preserves the generalist abilities of LLMs in various domains and modalities. It outperforms existing work on language modeling and can be applicable to instruction-tuned LMs and multi-modal LMs. Our system implementation further translates the theoretical memory savings achieved by AWQ into **3.2-3.3×** measured speedups over the FP16 implementations from Huggingface on desktop and mobile GPUs, democratizing LLM deployment on the edge.

## Acknowledgements

We thank MIT AI Hardware Program, National Science Foundation, NVIDIA Academic Partnership Award, MIT-IBM Watson AI Lab, Amazon and MIT Science Hub, Qualcomm Innovation Fellowship, Microsoft Turing Academic Program for supporting this research.

## References

- [1] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems*, 35:23716–23736, 2022.
- [2] Anas Awadalla, Irena Gao, Joshua Gardner, Jack Hessel, Yusuf Hanafy, Wanrong Zhu, Kalyani Marathe, Yonatan Bitton, Samir Gadre, Jenia Jitsev, Simon Kornblith, Pang Wei Koh, Gabriel Ilharco, Mitchell Wortsman, and Ludwig Schmidt. Openflamingo, March 2023.

<sup>\*</sup><https://github.com/ggerganov/llama.cpp>

<sup>†</sup><https://github.com/NVIDIA/FasterTransformer>

- [3] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.
- [4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.
- [5] Xinlei Chen, Hao Fang, Tsung-Yi Lin, Ramakrishna Vedantam, Saurabh Gupta, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco captions: Data collection and evaluation server. *arXiv preprint arXiv:1504.00325*, 2015.
- [6] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%\* chatgpt quality, March 2023.
- [7] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.
- [8] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.
- [9] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.
- [10] Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. *arXiv preprint arXiv:2212.09720*, 2022.
- [11] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.
- [12] Steven K Esser, Jeffrey L McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S Modha. Learned step size quantization. *arXiv preprint arXiv:1902.08153*, 2019.
- [13] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.
- [14] Elias Frantar, Saleh Ashkboos, Torsten Hoefer, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.
- [15] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.
- [16] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.
- [17] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *ICLR*, 2016.
- [18] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.
- [19] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.
- [20] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *Machine Learning: ECML 2004: 15th European Conference on Machine Learning, Pisa, Italy, September 20-24, 2004. Proceedings 15*, pages 217–226. Springer, 2004.
- [21] Jing Yu Koh, Ruslan Salakhutdinov, and Daniel Fried. Grounding language models to images for multi-modal generation. *arXiv preprint arXiv:2301.13823*, 2023.
- [22] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. *arXiv preprint arXiv:2301.12597*, 2023.
- [23] Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. Brecq: Pushing the limit of post-training quantization by block reconstruction. *arXiv preprint arXiv:2102.05426*, 2021.

- [24] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022.
- [25] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems*, 33:11711–11722, 2020.
- [26] Haotian Liu, Chunyuan Li, Qingsong Wu, and Yong Jae Lee. Visual instruction tuning. 2023.
- [27] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.
- [28] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or down? adaptive rounding for post-training quantization. In *International Conference on Machine Learning*, pages 7197–7206. PMLR, 2020.
- [29] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1325–1334, 2019.
- [30] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021.
- [31] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.
- [32] Gunho Park, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. nuqmm: Quantized matmul for efficient inference of large-scale generative language models. *arXiv preprint arXiv:2206.09557*, 2022.
- [33] Victor Sanh, Albert Webson, Colin Raffel, Stephen H Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. Multitask prompted training enables zero-shot task generalization. *arXiv preprint arXiv:2110.08207*, 2021.
- [34] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.
- [35] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. High-throughput generative inference of large language models with a single gpu. *arXiv preprint arXiv:2303.06865*, 2023.
- [36] Rohan Taori, Ishaaq Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. [https://github.com/tatsu-lab/stanford\\_alpaca](https://github.com/tatsu-lab/stanford_alpaca), 2023.
- [37] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.
- [38] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- [39] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.
- [40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [41] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-Aware Automated Quantization with Mixed Precision. In *CVPR*, 2019.
- [42] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.
- [43] Xiuying Wei, Yunchen Zhang, Yuhang Li, Xiangguo Zhang, Ruihao Gong, Jinyang Guo, and Xianglong Liu. Outlier suppression+: Accurate quantization of large language models by equivalent and optimal shifting and scaling. *arXiv preprint arXiv:2304.09145*, 2023.
- [44] Xiuying Wei, Yunchen Zhang, Xiangguo Zhang, Ruihao Gong, Shanghang Zhang, Qi Zhang, Fengwei Yu, and Xianglong Liu. Outlier suppression: Pushing the limit of low-bit transformer language models. *arXiv preprint arXiv:2209.13325*, 2022.

- [45] Xiuying Wei, Yunchen Zhang, Xiangguo Zhang, Ruihao Gong, Shanghang Zhang, Qi Zhang, Fengwei Yu, and Xianglong Liu. Outlier suppression: Pushing the limit of low-bit transformer language models, 2022.
- [46] Guangxuan Xiao, Ji Lin, Mickael Seznec, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*, 2022.
- [47] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers, 2022.
- [48] Renrui Zhang, Jiaming Han, Aojun Zhou, Xiangfei Hu, Shilin Yan, Pan Lu, Hongsheng Li, Peng Gao, and Yu Qiao. Llama-adapter: Efficient fine-tuning of language models with zero-init attention. *arXiv preprint arXiv:2303.16199*, 2023.
- [49] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuhui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

## A Broader Impacts and Limitations

**Broader impacts.** In this paper, we propose a general technique to enable accurate and efficient low-bit weight-only quantization of large language models (LLMs). It makes LLMs more efficient and accessible and thus may inherit the impacts of LLMs. On the positive side, quantization helps to democratize LLMs, which helps to benefit more people (especially those with lower income). It reduces the costs and hardware barrier of deploying LLMs and facilitates edge inference of these models, addressing the data privacy issue (since we no longer need to send data to the cloud). On the negative side, LLMs may be exploited by malicious users to produce misinformation and manipulation. Quantization can not prevent such negative effects but it does not make it worse.

**Limitations.** In this paper, we follow previous work [9, 14, 46, 47, 10] to mostly benchmark the quantized models on standard accuracy metrics like perplexity and accuracy. However, besides accuracy, there are other important metrics for LLM benchmark like robustness, fairness, bias, toxicity, helpfulness, calibration, *etc.* [24]. We think it would be helpful to perform a more holistic evaluation of quantized LLMs covering these aspects, which we leave to future work. Furthermore, we only study low-bit integer quantization of LLMs due to easier data type casting on hardware. There might be a further improvement from changing data types (*e.g.*, FP4 [10]), which we do not include in the study.

## B Amount of Computation

We study the post-training quantization (PTQ) of LLMs in this work. The computation requirement is generally modest since we do not rely on any backpropagation. We used one NVIDIA A100 GPU for smaller models (<40B parameters) and 2-4 A100 GPUs for larger models due to memory limits.

The quantization process is generally fast, requiring a few GPU hours (ranging from 0.1 to 3, depending on the model size). The accuracy measurement time depends on the model and dataset sizes: testing LLaMA-65B (the biggest model we tested on multiple datasets) on 4 common sense QA tasks requires 3 GPU hours; testing it on MMLU (consisting of 57 sub-datasets) requires 5 GPU hours. The GPU hours would be smaller for smaller models and datasets (*e.g.*, WikiText-2).

## C Limitation with No-group Quantization

Our method searches for good scaling to protect the salient weight channels. It works pretty well under grouped quantization, matching the same accuracy as keeping salient weights in FP16 (Figure 1). However, such a scaling-based method can only protect *one* salient channel for *each group*. It is not a problem for grouped quantization (we only need to protect 0.1%-1% of salient channels, the group size is usually small, like 128, so we need to protect fewer than 1 channel in each group on average). But for no-group quantization, we can only protect one input channel for the *entire weight*, which may not be enough to bridge the performance degradation. As shown in Table 7, under INT3-g128 quantization, AWQ achieves similar performance compared to keeping 1% salient weights in FP16. While under INT3 no-group quantization, there is still a noticeable gap. Nonetheless, we want to stress that the performance of no-group quantization is still far behind grouped quantization at a similar cost. Therefore, grouped quantization is a *more practical solution* for LLM compression for edge deployment and AWQ can effectively improve the quantized performance under this setting.

PPL ↓	FP16	INT3 (group 128)			INT3 (no group)		
		RTN	1% FP16	AWQ	RTN	1% FP16	AWQ
OPT-6.7B	12.29	43.16	13.02	12.99	21160	14.67	18.11
LLaMA-7B	9.49	12.10	10.77	10.82	50.45	14.06	20.52

**Table 7.** AWQ can match the performance of keeping 1% salient weights in FP16 under grouped quantization without introducing mixed-precisions, but not for no-group quantization. Nonetheless, grouped quantization has a far better performance compared to no-group, making it a far more practical setting for weight-only quantization of LLMs, while AWQ performs quite well under this setting. Results are perplexity on the WikiText-2 dataset.

# ByteTransformer: A High-Performance Transformer Boosted for Variable-Length Inputs

Yujia Zhai,<sup>\*</sup>

Chengquan Jiang,<sup>†¶</sup> Leyuan Wang,<sup>†</sup> Xiaoying Jia,<sup>†</sup> Shang Zhang,<sup>‡</sup>

Zizhong Chen,<sup>\*</sup> Xin Liu,<sup>†§</sup> Yibo Zhu<sup>†</sup>

<sup>\*</sup>University of California, Riverside

<sup>†</sup>ByteDance Ltd.

<sup>‡</sup>NVIDIA Corporation

<sup>§</sup>Correspondence to liuxin.ai@bytedance.com

<sup>¶</sup>These authors contributed equally to this work.

**Abstract**—Transformers have become keystone models in natural language processing over the past decade. They have achieved great popularity in deep learning applications, but the increasing sizes of the parameter spaces required by transformer models generate a commensurate need to accelerate performance. Natural language processing problems are also routinely faced with variable-length sequences, as word counts commonly vary among sentences. Existing deep learning frameworks pad variable-length sequences to a maximal length, which adds significant memory and computational overhead. In this paper, we present ByteTransformer, a high-performance transformer boosted for variable-length inputs. We propose a padding-free algorithm that liberates the entire transformer from redundant computations on zero padded tokens. In addition to algorithmic-level optimization, we provide architecture-aware optimizations for transformer functional modules, especially the performance-critical algorithm Multi-Head Attention (MHA). Experimental results on an NVIDIA A100 GPU with variable-length sequence inputs validate that our fused MHA outperforms PyTorch by 6.13x. The end-to-end performance of ByteTransformer for a forward BERT transformer surpasses state-of-the-art transformer frameworks, such as PyTorch JIT, TensorFlow XLA, Tencent TurboTransformer, Microsoft DeepSpeed-Inference and NVIDIA FasterTransformer, by 87%, 131%, 138%, 74% and 55%, respectively. We also demonstrate the general applicability of our optimization methods to other BERT-like models, including ALBERT, DistilBERT, and DeBERTa.

**Index Terms**—Transformer, BERT, Multi-head Attention, MHA, Natural Language Processing, NVIDIA GPU, CUTLASS

## I. INTRODUCTION

The transformer model [1] is a proven effective architecture widely used in a variety of deep learning (DL) applications, such as language modeling [2], [3], neural machine translation [1], [4] and recommendation systems [5], [6]. The last decade has witnessed rapid developments in natural language processing (NLP) pre-training models based on the transformer model, such as Seq2seq [1], GPT-2 [7] and XLNET [3], which have also greatly accelerated the progress of NLP. Of all the pre-training models based on transformers, Bidirectional Encoder Representations from Transformers (BERT), proposed in 2018 [2], is arguably the most seminal, inspiring a series of subse-

We have made ByteTransformer open-source and available at a public GitHub repository: <https://github.com/bytedance/ByteTransformer>.

quent works and outperforming reference models on a dozen NLP tasks at the time of creation.

BERT-like models consume increasingly larger parameter space and correspondingly more computational resources. When BERT was discovered, a large model required 340 million parameters [8], but currently a full GPT-3 model requires 170 billion parameters [9]. The base BERT model requires 6.9 billion floating-point operations to inference a 40-word sentence, and this number increases to 20 billion when translating a 20-word sentence using a base Seq2Seq model [10]. The size of the parameter space and the computational demands increase the cost of the training and inference for BERT-like models, which requires the attention of the DL community in order to accelerate these models.

To exploit hardware efficiency, DL frameworks adopt a batching strategy, where multiple batches are executed concurrently. Since batched execution requires task shapes in different batches to be identical, DL frameworks presume fixed-length inputs when designing the software [11]–[14]. However, this assumption cannot always hold, because transformer models are often faced with variable-length input problems [8], [10]. In order to deploy models with variable-length inputs directly to conventional frameworks that support only fixed-length models, a straightforward solution is to pad all sequences with zeros to the maximal sequence length. However, this immediately brings in redundant computations on wasted padded tokens. These padded zeros also introduce significant memory overhead that can hinder a large transformer model from being efficiently deployed.

Existing popular DL frameworks, such as Google TensorFlow with XLA [15], [16], Meta PyTorch with JIT [17], and OctoML TVM [18], leverage the domain-specific just-in-time compilation technique to boost performance. Another widely-adopted strategy to generate low-level performance optimization is delicate manual tuning: NVIDIA TensorRT [19], a DL runtime, falls into this category. Yet all of these frameworks require the input sequence lengths to be identical to exploit the speedup of batch processing. To lift the restriction on fixed sequence lengths, Tencent [10] and Baidu [8] provide explicit support for models with variable sequence lengths. They group sequences with similar lengths before launching

batched kernels to minimize the padding overhead. However, this proactive grouping approach still introduces irremovable padding overhead when grouping and padding sequences with similar yet different lengths.

In contrast to training processes that can be computed offline, the inference stage of a serving system must be processed online with low latency, which imposes high performance requirements on DL frameworks. A highly efficient DL inference framework for NLP models requires delicate kernel-level optimizations and explicit end-to-end designs to avoid wasted computations on zero tokens when handling variable-length inputs. However, existing DL frameworks do not meet these expectations. In order to remedy this deficit, we present ByteTransformer, a highly efficient transformer framework optimized for variable-length inputs in NLP problems. We not only design an algorithm that frees the entire transformer of padding when dealing with variable-length sequences, but also provide a set of hand-tuned fused GPU kernels to minimize the cost of accessing GPU global memory. More specifically, our contributions include:

- We design and develop ByteTransformer, a high-performance GPU-accelerated transformer optimized for variable-length inputs. ByteTransformer has been deployed to serve world-class applications including TikTok and Douyin of ByteDance.
- We propose a padding-free algorithm that packs the input tensor with variable-length sequences and calculates the positioning offset vector for all transformer operations to index, which keeps the whole transformer pipeline free from padding and calculations on zero tokens.
- We propose a fused Multi-Head Attention (MHA) to alleviate the memory overhead of the intermediate matrix, which is quadratic to the sequence length, in MHA without introducing redundant calculations due to padding for variable-length inputs. Part of our fused MHA has been deployed in the production code base of NVIDIA CUTLASS.
- We hand-tune the memory footprints of layer normalization, adding bias and activation to squeeze the final performance of the system.
- We benchmark the performance of ByteTransformer on an NVIDIA A100 GPU for forward pass of BERT-like transformers, including BERT, ALBERT, DistilBERT, and DeBERTa. Experimental results demonstrate our fused MHA outperforms standard PyTorch attention by 6.13X. Regarding the end-to-end performance of standard BERT transformer, ByteTransformer surpasses PyTorch, TensorFlow, Tencent TurboTransformer, Microsoft DeepSpeed and NVIDIA FasterTransformer by 87%, 131%, 138%, 74%, and 55%, respectively.

The rest of the paper is organized as follows: we introduce background and related works in Section II, and then detail our systematic optimization approach in Section III. Evaluation results are given in Section IV. We conclude our paper and present future work in Section V.

## II. BACKGROUND AND RELATED WORKS

We provide an overview of the transformer model, including its encoder-decoder architecture and multi-head attention layer. We also survey related works on DL framework acceleration.

### A. The transformer architecture

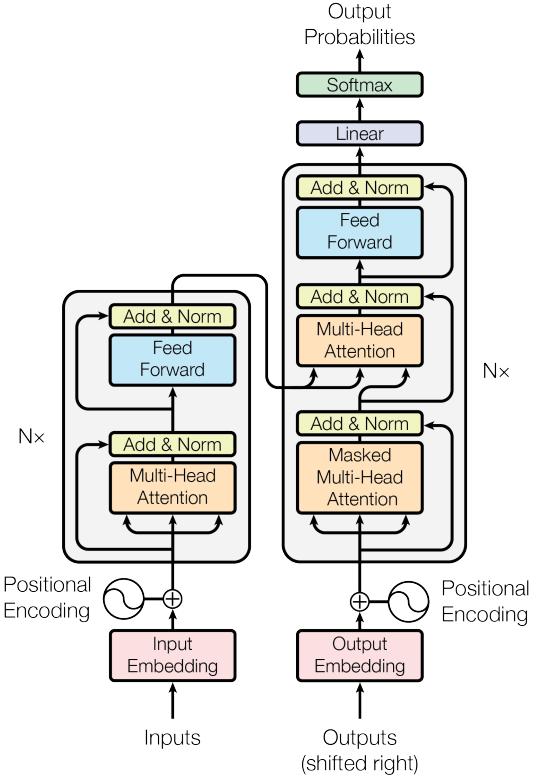


Fig. 1: The transformer architecture. [1]

Figure 1 shows the encoder-decoder model architecture of the transformer. It consists of stacks of multiple encoder and decoder layers. In an encoder layer, there is a multi-head attention layer followed by a feed-forward network (FFN) layer. A layer normalization (layernorm) operation is applied after both MHA and FFN. In a decoder layer, there are two sets of consecutive MHA layers and one FFN layer, and each operation is normalized with a layernorm. The FFN is used to improve the capacity of the model. In practice, FFN is implemented by multiplying the tensor by a larger scaled tensor using GEMM. Here we skip the embedding descriptions in the figure, and refer an interested reader to [1] for details. Although we show both encoder and decoder modules for this transformer, a BERT transformer model only contains the encoder section [2]. In this paper, we present optimizations for BERT-like transformer models, which can be extended to other transformers containing decoder sections.

Self-attention is a key module of the transformer architecture. Conceptually, self-attention computes the significance of each position of the input sequence, with the information from other positions considered. A self-attention receives three input tensors: query (Q), key (K), and value (V). Self-attention can

be split into multiple heads. The Q and K tensors are first multiplied ( $1^{st}$  GEMM) to compute the dot product of the query against all keys. This dot product is then scaled by the hidden dimension  $d_k$  and passed through a softmax function to calculate the weights corresponding to the value tensor. Each head of the output tensor is concatenated before going through another linear layer by multiplying against tensor V ( $2^{nd}$  GEMM). Expressing self-attention as a mathematical formula, we have:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \times V \quad (1)$$

Whereas the formula of multi-head attention is:  $\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)$ , here  $\text{head}_i = \text{Attention}(Q_i, K_i, V_i)$ .

### B. Related works on DL acceleration

Performance is a crucial aspect in the real-world deployment of software systems, attracting significant attention across various applications [20]–[22], including DL frameworks. The conventional DL frameworks, such as PyTorch, TensorFlow, TVM, and TensorRT are designed explicitly for fixed-length input tensors. When dealing with NLP problems with variable-length input, all sequences are padded to the maximal length, which leads to significant wasted calculations on zero tokens. A few DL frameworks, such as Tencent TurboTransformer [10] and NVIDIA FasterTransformer [23], employ explicit designs for variable-length inputs. TurboTransformer designs run-time algorithms to group and pad sequences with similar lengths to minimize the padding overhead. TurboTransformer also uses a run-time memory scheduling strategy to improve end-to-end performance. Kernel-level optimizations are of the same significance as algorithmic optimizations. NVIDIA’s FasterTransformer uses vendor-specific libraries such as TensorRT and cuBLAS [24] as its back-end, which provide optimized implementations of various operations at the kernel level.

Other end-to-end DL frameworks have also presented optimizations for BERT-like transformers, such as E.T. [25] and DeepSpeed-Inference [26]. E.T. introduces a novel MHA architecture for NVIDIA Volta GPUs and includes pruning designs for end-to-end transformer models. In contrast, ByteTransformer targets unpruned models and is optimized for NVIDIA Ampere GPUs. DeepSpeed-Inference is optimized for large distributed models on multiple GPUs, while ByteTransformer currently focuses on lighter single-GPU models.

In addition to end-to-end performance acceleration, the research community has also made focused efforts to improve a key algorithm of the transformer, multi-head attention. PyTorch provides a standard implementation of MHA [27]. NVIDIA TensorRT utilizes a fused MHA for short sequences with lengths up to 512, as described in [28]. To handle longer sequences, FlashAttention was proposed by Stanford researchers in [29]. FlashAttention assigns the workload of a whole attention unit to a single threadblock (CTA). However, this approach can result in underutilization on wide GPUs

when there are not enough attention units assigned. Our fused MHA, on the other hand, provides high performance for both short and long sequences for variable-length inputs without leading to performance degradation in small-batch scenarios.

TABLE I. Summarizing state-of-the-art transformers.

	variable-len support	kernel tuning	fused MHA	kernel fusion
Tensorflow XLA	no	yes	no	no
PyTorch JIT	no	yes	no	no
FasterTransformer	yes	yes	$\leq 512$	no
TurboTransformer	yes	yes	no	partially
<b>ByteTransformer</b>	yes	yes	yes	yes

Table I surveys state-of-the-art transformers. TensorFlow and PyTorch provide tuned kernels but require padding for variable-length inputs. NVIDIA FasterTransformer and Tencent TurboTransformer, although providing support for variable-length inputs, do not perform comprehensive kernel fusion or explicit optimization for the hot-spot algorithm MHA for any length of sequence. In addition, TurboTransformer only optimizes part of the fusible operations in the transformer model, such as layernorm and activation, namely ‘partial kernel fusion’ in the table. Our ByteTransformer, in contrast, starting with a systemic profiling to locate bottleneck algorithms, precisely tunes a series of kernels including the key algorithm MHA. We also propose a padding-free algorithm which completely removes redundant calculations for variable-length inputs from the entire transformer.

## III. DESIGNS AND OPTIMIZATIONS

In this section, we present our algorithmic and kernel-level optimizations to improve the end-to-end performance of BERT transformer under variable-length inputs.

### A. Math expression of BERT transformer encoder

Figure 2(a) illustrates the architecture of the transformer encoder. The input tensor is first processed through the BERT pipeline, where it is multiplied by a built-in attribute matrix to perform Q, K, and V positioning encoding. This operation can be implemented using three separate GEMM operations or in batch mode. Realizing that the corresponding attribute matrices to Q, K, and V are all the same shape (hidden\_dim x hidden\_dim), we pack them to continuous memory space and launch a single batched GEMM kernel that calculates Q, K, and V to reduce the kernel launch overhead at runtime. Bias matrices for Q, K, and V are then added to the encoded tensor, which is passed through the self-attention module. In addition to the multi-head attention module, the BERT transformer encoder includes projection, feed forward network, and layer normalization. The encoder pipeline can be represented as a series of mathematical operations, including six GEMMs (shown in light purple) and other memory-bound operations (shown in light blue).

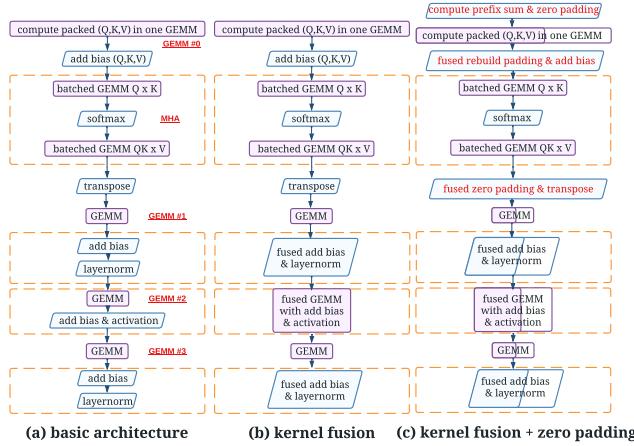


Fig. 2: BERT transformer architecture and optimizations.

### B. Profiling for single-layer standard BERT transformer

We implement the pipeline of Figure 2 (a) by calling cuBLAS and profile its single-layer performance on an NVIDIA A100 GPU. We adopt the standard BERT transformer configuration (batch size: 16, head number: 12, head size: 64) and profile for two different sequence lengths: 256 and 1024.

Figure 3 shows the performance breakdown for two sequence lengths. GEMM0 to GEMM3 refer to the consecutive four GEMMs that are enumerated from GEMM #0 to GEMM #3 in Figure 2 (a). The other two batched GEMMs are part of the attention module and are therefore profiled together with the softmax as a whole, referred to as MHA in Figure 3. The two sets of "add bias and layernorm" operations are referred to as layernorm0 and layernorm1. The profiling results show that the compute-bound GEMM operations account for 61% and 40% of the total execution time for both test cases. The attention module, which includes a softmax and two batched GEMMs, is the most time-consuming part of the transformer. As the sequence length increases to that of a GPT-2 model (1024), attention accounts for 49% of the total execution time, while the remaining memory-bound operations (layernorm, add bias and activation) only take up 11%-17%.

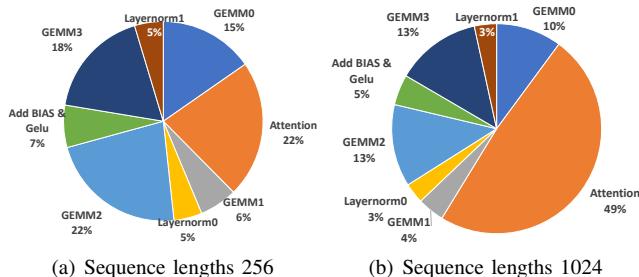


Fig. 3: Performance breakdown of forward BERT transformer.

### C. Fusing memory-bound operations of BERT transformer

Since cuBLAS uses architectural-aware optimizations for high performance GEMMs, presumably there remain limited opportunities for further acceleration. Therefore, we turn our eyes to optimizing the modules containing memory-bound operations, such as attention (with softmax), feed forward network (with layernorm) and add bias followed by element-wise activation. We improve these operations by fusing distinct kernels and reusing data in registers to reduce global memory access. Figure 2 (b) presents the BERT transformer pipeline with memory-bound kernel fusion, where we fuse layernorm and activation with their consecutive kernels.

1) *Add bias and layer normalization:* These operations account for 10% and 6% of the overall execution time for sequence lengths 256 and 1024, respectively. After MHA, the result tensor (`valid_word_cnt` × `hidden_dim`) needs to first be added upon the input tensor (bias) and perform layer normalization. Here hidden dimension (`hidden_dim`) equals `head_num` × `head_size`. In standard BERT configuration, head number and head size are fixed to 12 and 64. The naive implementation introduces two rounds of memory access to load and store the tensor. We provide a fused kernel that only needs to access the global memory in one round to finish both layernorm and adding bias. Kernel fusion for this sub-kernel improves the performance by 61%, which accordingly increases the single-layer BERT transformer performance by 3.2% for sequence lengths ranging 128 to 1024 in average.

2) *add bias and activation:* These operations account for 7% and 5% of the overall execution time for sequence lengths 256 and 1024, respectively. After the projection via matrix multiplication, the result tensor will be added against the input tensor and perform an element-wise activation using GELU [30]. Our fused implementation, rather than storing the GEMM output to global memory and loading it again to conduct adding bias and activation, re-uses the GEMM result matrix at the register level by implementing a customized and fused CUTLASS [31] epilogue. Experimental results validate that our fused GEMM perfectly hides the memory latency of bias and GELU into GEMM. After this step, we further improve the single-layer BERT transformer by 3.8%.

### D. The zero padding algorithm for variable-length inputs

Because the real-time serving process receives sentences with various words as input tensor, the sequence lengths can often be different among batches. For such an input tensor composed of sentences with variable lengths, the conventional solution is to pad them to the maximal sequence length with useless tokens, which leads to significant computational and memory overhead. In order to address this issue, we propose the zero padding algorithm to pack the input tensor and store the positioning information for other transformer operations to index the original sequences.

Figure 4 presents the details of the zero padding algorithm. We use an input tensor with 3 sentences (proceeded in 3 batches) as an example. The longest sentence contains 5 word tokens while the other two have 2 and 4 words. The height

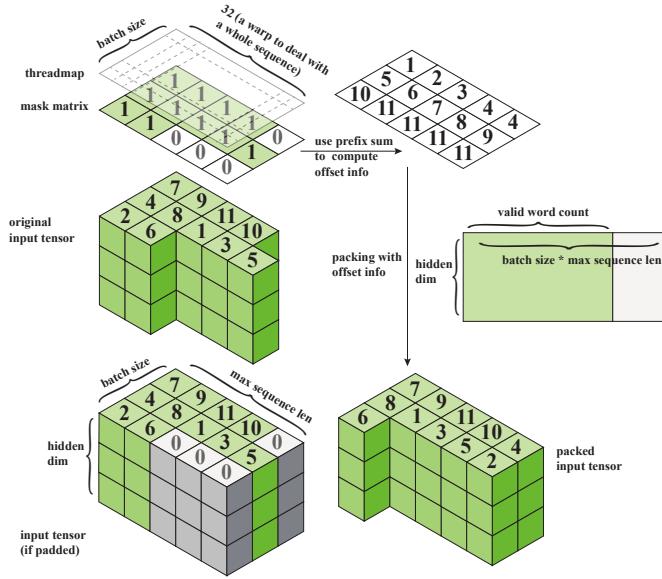


Fig. 4: The zero padding algorithm.

of the sample input tensor is 3, which is equal to the hidden dimension. The conventional method is to pad all sentences to the maximal sequence length by filling zeros. The elements, either 1 or 0, of the mask matrix correspond respectively to a valid token or a padded token of an input tensor with variable size. By calculating the prefix sum of the mask matrix, we can skip the padded tokens and provide the position indices of all valid tokens. We implement an efficient CUDA kernel to calculate the prefix sum and the position offset. Each warp computes the prefix sum for tokens of a whole sentence, so in total there are `batch_size` warps assigned in each threadblock for prefix sum calculation. Once the prefix sum is computed, we pack the input tensor to a continuous memory area so that the total number of words used in future calculations is reduced from `seq_len` × `batch_size` to the actual valid word count of the packed tensor.

Figure 2 (c) presents the detailed modifications on BERT by introducing our zero padding algorithm. Before conducting the positioning encoding, we calculate the prefix sum of the mask matrix to pack the input tensor so that we avoid computations on useless tokens in the first GEMM. Since batched GEMM in MHA requires identical problem shapes among different batches, we unpack the tensor before entering the attention module. Once MHA is completed, we pack the tensor again such that all remaining operations can benefit from the zero padding algorithm. The final result tensors are validated element-by-element against TensorFlow such that the correctness and accuracy are ensured. It is worth mentioning that padding and remove padding operations are fused with existing memory-bound footprints such as adding bias and transpose to minimize the overhead led by this feature.

Our presented padding-free algorithm is designed to ensure semantic preservation. We maintain an array that stores the mapping relationship of the valid tokens between the original

tensor and the packed tensor. The transformer operates on the packed tensor, and intermediate operations, such as MHA, layernorm and activation, refer to this position array to ensure the correctness. At the end of each layer, we reconstruct the output tensor according to the position array such that the whole pipeline is semantic preserving.

	Baseline	Zero Padding	Zero Padding + fused MHA
GEMM0	$6mk^2$	$6(\alpha \cdot m)k^2$	$6(\alpha \cdot m)k^2$
MHA	$4\frac{m^2}{bs}k$	$4\frac{m^2}{bs}k$	$4\frac{(\alpha \cdot m)^2}{bs}k$
GEMM1	$2mk^2$	$2(\alpha \cdot m)k^2$	$2(\alpha \cdot m)k^2$
GEMM2	$8mk^2$	$8(\alpha \cdot m)k^2$	$8(\alpha \cdot m)k^2$
GEMM3	$8mk^2$	$8(\alpha \cdot m)k^2$	$8(\alpha \cdot m)k^2$

TABLE II. The computation number needed for variable-length inputs, where average sequence length =  $\alpha$  \* maximum,  $m$  denotes `batch_size` · `max_seq_len`,  $k$  is denote hidden dimension `head_num` · `head_size`,  $bs$  denotes the batch size.

Table II counts the floating point computations of a single-layer BERT transformer. The computations of memory-bound operations are not included since they are negligible compared with the listed modules. Enabling the zero padding algorithm eliminates redundant computations for all compute-intensive modules other than MHA due to the restrictions of batched GEMM. When the average sequence length is equal to 60% of the maximum, turning on the zero padding algorithm further accelerates the BERT transformer by 24.7%.

#### E. Optimizing multi-head attention

The zero-padding algorithm, although it effectively reduces wasted calculations for variable-length inputs, cannot directly benefit batched GEMM operations in MHA. This disadvantage becomes increasingly significant when the sequence length increases, as demonstrated in Table II. The complexity of MHA is quadratic to the sequence length, while the complexity of all other GEMMs is linear to the sequence length. This motivates us to provide a high-performance fused MHA while maintaining the benefits of the zero-padding algorithm. With our fused MHA, attention no longer faces redundant calculations on useless tokens, as shown in Table II.

1) *Unpadded fused MHA for short sequences*: For short input sequences, we hold the intermediate matrix in shared memory and registers throughout the MHA computation kernel to fully eliminate the quadratic memory overhead. We also access Q, K, and V tensors according to the positioning information obtained in the prefix sum calculation step to avoid redundant calculations on padding zeros for the MHA module.

Algorithm III.1 shows the pseudo code of our fused MHA for short sequences. We launch a 3-dimensional grid map: {`head_num`, `seq_len`/`split_seq_len`, `batch_size`}. Here `split_seq_len` is a user-defined parameter to determine the size of a sequence tile preceded by a threadblock (typically set to 32 or 48). The warp count of a threadblock is computed by the maximal sequence length: `split_seq_len`/16 × (`seq_len`/16). Each threadblock loads a chunk of Q (`split_seq_len` × `head_size`), K (`max_seq_len` × `head_size`) and V ((`head_size` ×

### Algorithm III.1: Unpadded fused MHA for short sequences

```

1 /* define skew offset to avoid bank conflict */
2 #define SKEW_HALF 8
3 Shared memory:
4 __half s_kv [max_seq_len][size_per_head + SKEW_HALF];
5 __half s_query [split_seq_len][size_per_head +
    SKEW_HALF];
6 __half s_logits [max_seq_len][size_per_head +
    SKEW_HALF];
7 /* warps collaboratively fill s_query with adding bias fused */
8 Load __half2 q_bias
9 for seq_id = warp_id : warp_num : split_seq_len do
10     query = Q[batch_seq_offset + seq_id +
        thread_offset];
11     offset = seq_id*(head_size+SKEW_HALF)+(lane_id*2);
12     (__half2 *)s_query[offset] = fast_add(query,
        k_bias);
13 /* warps collaboratively fill s_kv with adding bias fused */
14 Load __half2 k_bias
15 for seq_id = warp_id : warp_num : batch_seq_len do
16     key = K[batch_seq_offset + seq_id +
        thread_offset];
17     offset = seq_id*(head_size+SKEW_HALF)+(lane_id*2);
18     (__half2 *)s_kv[offset] = fast_add(key, k_bias);
19 /* compute Q*K using WMMA */
20 Clear wmma fragment QK to zero
21 for k_id = 0 : head_size / 16 do
22     Load 16x16 wmma fragments of Q
23     Load 16x16 wmma fragments of K
24     Update QK = Q * K + QK using wmma::mma_sync
25 Store fragment QK to s_logits using wmma::store_matrix_sync
26 /* Compute softmax */
27 for seq_id = warp_id : warp_num : batch_seq_len do
28     float logits[max_seq_len];
29     each thread loads a whole sequence to fill local registers
30     /* 1st round of reduction with register-level data re-use*/
31     compute max_val in local registers
32     /* register-level data re-use*/
33     compute P = exp(P - max) and update local registers
34     /* 2st round of reduction with register-level data re-use*/
35     compute sum_val in local registers
36     /* register-level data re-use*/
37     compute P = P/sum_val and stream to s_logits
38 /* warps collaboratively fill s_kv with adding bias fused */
39 Load __half2 v_bias
40 for seq_id = warp_id : warp_num : batch_seq_len do
41     value = V[batch_seq_offset + seq_id +
        thread_offset];
42     offset = seq_id*(head_size+SKEW_HALF)+(lane_id*2);
43     (__half2 *)s_kv[offset] = fast_add(value, v_bias);
44 /* Similar to Q * K so omitting the details here */
45 Compute P * V using wmma and stream to global memory

```

`max_seq_len))` into shared memory and computes MHA for a tile of the result tensor. We allocate three shared-memory buffers to hold  $Q$ ,  $K$ ,  $V$  sub-matrices. Due to the algorithmic nature of MHA, we can re-use  $K$  and  $V$  chunks in the same shared-memory buffer `s_kv`. The intermediate matrix of MHA is held and re-used in another pre-allocated shared-memory buffer `s_logits`.

The workflow of fused MHA for short sequences is straightforward yet efficient. Each thread first loads its own tile of  $Q$  and  $K$  into shared memory and computes GEMM for  $P = Q \times K$ . The element-wise adding bias and scaling operations are both fused with the load process to hide the memory latency. GEMM is computed using the CUDA `wmma` intrinsic to leverage tensor cores of NVIDIA Ampere GPUs. The intermediate matrix  $P$  is held in shared memory during

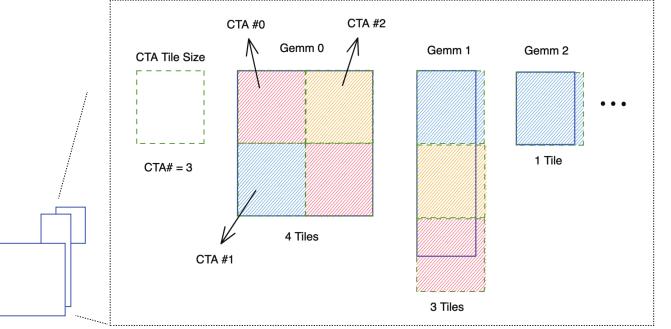


Fig. 5: Grouped GEMM demonstration.

the reduction. Because we explicitly design this algorithm for short sequences, each thread can load a whole sequence of  $P$  from shared memory into register files for both reduction and element-wise exponential transform in softmax. Once the softmax operation is completed, we load a  $K$  tile to shared memory to compute the second GEMM  $O = P \times V$ , and then store the result tensor  $O$  to the global memory.

2) *Unpadded fused MHA for long sequences:* Because of the limited resources of register files and shared memory, the previous fused MHA is no longer feasible for long sequences. Therefore, we set 384 to be the cut-off sequence length and propose a grouped GEMM based fused MHA for large models.

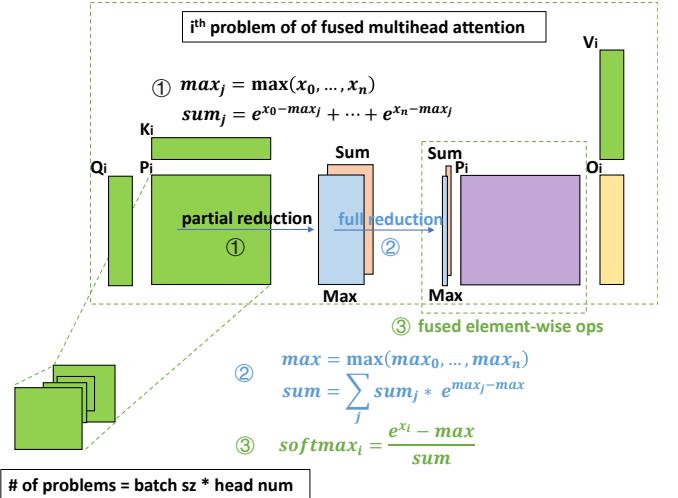


Fig. 6: Grouped-GEMM-based FMHA. The prototype of our fused MHA has been upstreamed to and released with CUTLASS 2.10. Source codes are available at [32].

The Grouped GEMM idea is first presented by NVIDIA CUTLASS [31]. Different from batched GEMM, where all GEMM sub-problems are required to have an identical shape, grouped GEMM allows arbitrary shapes for sub-problems. This is enabled by a built-in scheduler that iterates over all GEMM sub-problems in a round-robin manner. Figure 5 demonstrates the idea of grouped GEMM using an example with 3 sub-problems. Supposing 3 threadblocks (CTAs) are launched, each CTA calculates a fix-sized CTA tile at each

step until all GEMM sub-problems have been covered. GPU computes in waves, logically. In the first wave, All three CTAs calculate 3 tiles (light red, light yellow and light blue in the figure). And then in the second CTA wave, CTA #0 moves to the bottom-right tile of GEMM 0 while CTA #1 and CTA #2 move to sub-problems of GEMM 1. In the final CTA wave, CTA #0 and CTA #1 continue to compute tasks in GEMM 1 and GEMM 2 while CTA #2 keeps idle because there are no more available tiles in the computational graph.

Since grouped GEMM lifts the restriction on the shape of sub-problems, it can directly benefit MHA problems with variable-length inputs. Figure 6 presents our grouped-GEMM-based fused MHA for long sequences. The total number of MHA problems is equal to `batch_size × head_num`. The MHA problems among different batches have different sequence lengths, while sequence lengths within the same batch are identical. The grouped GEMM scheduler iterates over all attention units in a round-robin manner. In each attention unit, we first compute GEMM  $P_i = Q_i \times K_i$ , and conduct softmax on  $P_i$ . The second GEMM  $O_i = P_i \times V_i$  provides us with the final attention result. Here  $i$  indicates the  $i^{th}$  problem of grouped MHA with variable shapes. The softmax operation is fused with GEMMs to hide the memory latency. We have upstreamed the prototype of our grouped GEMM based fused MHA into NVIDIA CUTLASS [32].

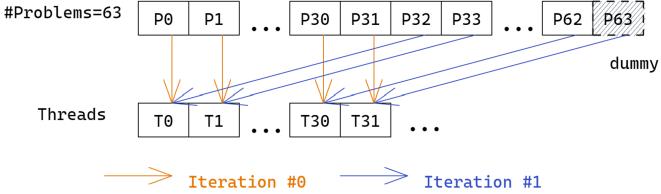


Fig. 7: Warp prefetching for grouped GEMM.

Grouped GEMM frequently checks with the built-in scheduler on the current task assignments, which leads to the runtime overhead. To address this issue, we propose an optimization over the built-in CUTLASS group GEMM scheduler. Figure 7 shows our optimization for the original CUTLASS grouped GEMM scheduler. Rather than asking one thread to compute the current tasks metadata, we have all 32 threads in a warp compute the tile indices to visit at one time. Therefore, we achieve 32X fewer scheduler visit overhead. In practice, this strategy brings a  $\sim 10\%$  improvement over the original CUTLASS grouped GEMM for standard BERT configurations. The prototype of this optimization has also been upstreamed to NVIDIA CUTLASS. We would refer an interested reader to [33] for detailed source codes.

In addition to optimizing the grouped GEMM scheduler, we fuse the memory footprints of softmax into two grouped GEMMs of MHA. Figure 8 shows the details of epilogue fusion for softmax reduction. A CTA computes an  $M_C \times N_C$  sub-matrix.  $M_C$  and  $N_C$  are both set to 128 to maximize the performance of GEMM. Under the default CUTLASS threadmap assignment, there are 128 threads per CTA, and

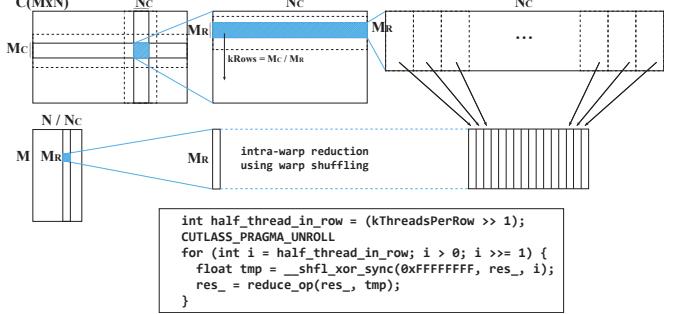


Fig. 8: Fused softmax reduction in grouped GEMM epilogue.

the threadmap is arranged as  $8 \times 16$ , where each thread holds a 128-bit register tile in each step. After the intra-thread reduction, the  $M_R \times N_C$  ( $8 \times 128$ ) sub-matrix is reduced to  $8 \times 16$ , with one reduced result held by one thread. We then conduct an intra-warp reduction to further reduce from the column dimension, which is implemented via CUDA warp shuffling for efficiency. Similar reductions (intra-thread followed by intra-warp reduction) are performed to compute both max and sum in epilogue. Once max and sum are both reduced, we store them to global memory.

The reduction in epilogue only provides us with partial reduction within a threadblock because cross-threadblock communication is impractical under the current CUDA programming model. Hence, we need to launch a separated lightweight kernel, as shown in Figure 6, to conduct the full reduction. In partial reduction, the target tensor of each attention unit is `seq_len × seq_len` while the full reduction just reduces a `seq_len × seq_len / 128`. Therefore, the workload of full reduction is negligible to that of partial reduction. In practice, the full reduction kernel only accounts for  $\sim 2\%$  of total execution time in fused MHA.

Once we have obtained the fully reduced `max` and `sum` vectors, we are ready to proceed element-wise transform  $\frac{e^{x_{ij} - \text{max}}}{\text{sum}}$  on the first GEMM's output matrix. To hide the memory latency, we fuse these element-wise operations into the mainloop of the second GEMM. Algorithm III.2 presents our modifications (marked in red) of the original CUTLASS GEMM mainloop to enable softmax fusion. The original GEMM mainloop adopts the pipelining strategy to alleviate memory access latencies on both global memory and shared memory. For shared memory accesses, double register tiles are utilized to ensure that what is consumed in the current iteration has always been loaded in the previous iteration. For global memory accesses, a multi-stage loading strategy is employed with the help of the `cp.async` instruction of NVIDIA Ampere GPUs. The `cp.async` instruction allows loading data asynchronously from global memory to shared memory without consuming registers. Multiple such transactions can be proceeded concurrently, and a stage barrier ensures selected stages to be synchronized. The number of load stages (`kStages`) is a compile-time constant defined by a user. Similar to shared memory accesses, loading from global

### Algorithm III.2: Mainloop fusion of grouped FMHA

```

1 Register Tiles:
2 WarpLoadedFragmentA warp_loaded_frag_A[2];
3 WarpLoadedFragmentB warp_loaded_frag_B[2];
4 WarpLoadedFragmentNormSum warp_loaded_frag_norm_sum;
5 Shared memory: (kStages + 1) shared-memory tiles for A and B
6 /* prologue */
7 Load k-invariant fused softmax tile to warp_loaded_frag_norm_sum
8 Prefetch kStages - 1 tiles of A to shared memory using cp.async
9 Prefetch kStages - 1 tiles of B to shared memory using cp.async
10 Prefetch a tile of A from shared memory to warp_loaded_frag_A[0]
11 Prefetch a tile of B from shared memory to warp_loaded_frag_B[0]
12 /* fused element-wise operation */
13 /* A = exp(A-max) */
14 elementwise_transform(
15   warp_loaded_frag_A[0],
16   warp_loaded_frag_norm_sum);
17 /* mainloop */
18 for k to -kStages + 1 do
19   /* Computes a warp-level GEMM */
20   /* with pipelined load during iterations */
21   for warp_mma_k = 0 to kWarpGemmIterations - 1 do
22     Prefetch warp_loaded_frag_A[(warp_mma_k + 1) % 2]
23     Prefetch warp_loaded_frag_B[(warp_mma_k + 1) % 2]
24     /* fused element-wise transform */
25     elementwise_transform(
26       warp_loaded_frag_A[(warp_mma_k + 1) % 2],
27       warp_loaded_frag_norm_sum);
28     /* Computes a warp-level GEMM*/
29     /* on data loaded in previous iteration */
30     warp_mma(
31       accum,
32       warp_loaded_frag_A[warp_mma_k % 2],
33       warp_loaded_frag_B[warp_mma_k % 2],
34       accum);
35     Prefetch a tile of A to shared memory using cp.async
36     Prefetch a tile of B to shared memory using cp.async

```

memory is also pipelined to overlap memory latency with computation. Therefore, kStages pieces of shared memory buffers are needed under the multi-stage pipeline scheme. As shown in Algorithm III.2, we preload the k-invariant vectors *sum* and *max* in prologue, and conduct element-wise transform right after the matrix elements are loaded into registers. Since the fused vectors are loaded outside of the GEMM mainloop, only negligible overhead is brought into the baseline GEMM and the memory latency to perform element-wise transform is perfectly hidden with GEMM computations.

The baseline MHA is a computational chain containing a batched GEMM, a softmax, and another batched GEMM. The time and memory complexity of all these operations are quadratic in the sequence length. Because the padding-free algorithm directly reduces the effective sequence length, MHA with variable-length input also gains a direct improvement. Our fused MHA, which is explicitly designed to handle both short and long sequences, incorporates the padding-free algorithm to alleviate the memory overhead of the intermediate matrix in MHA caused by padding for variable-length inputs. Our highly optimized MHA outperforms the standard PyTorch MHA by 6.13X and further accelerates the single-layer BERT transformer by 19% compared to the previous step. As a result, this fully optimized version surpasses the baseline implementation in Figure 2 (a) by 60%. Since the remaining operations of a forward BERT transformer are all near-optimal GEMM operations, we conclude our optimizations at this step.

### IV. EVALUATION

We evaluate our optimizations on an NVIDIA A100 GPU. The GPU device is connected to a node with four 32-core Intel Xeon Platinum 8336C CPUs, whose boost frequency is up to 4.00 GHz. The associated CPU main memory system has a capacity of 2TB at 3200 MHz. We compile programs using CUDA 11.6u2 with the optimization flag `O3`. We compare the performance of ByteTransformer with latest versions of state-of-the-art transformers, such as TensorFlow 2.8, PyTorch 1.13, Tencent TurboTransformer 0.5.1, Microsoft DeepSpeed-Inference 0.7.7, and NVIDIA FasterTransformer 5.1. All the tensors benchmarked in this paper, unless specified, are in the half-precision floating-point format (FP16) to leverage tensor cores of NVIDIA GPUs. The variable sequence lengths in this section are generated randomly based on a uniform distribution with a range from 1 to the maximum length. We average the reported performance data over tens of runs to minimize fluctuations.

#### A. Kernel fusion for layernorm and add-bias operations

As depicted in Figure 2, BERT transformer is composed of a series of GEMM and memory-bound operations. Since GEMM are accelerated by near-optimal vendor’s libraries cuBLAS and CUTLASS, we focus on optimizing the functional modules that involve memory-bound operations.

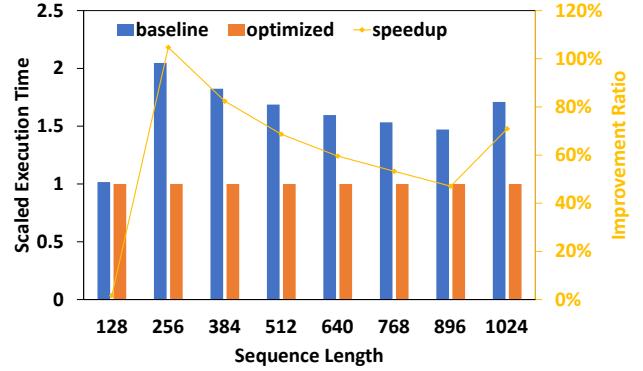


Fig. 9: Kernel fusion for add-bias and layernorm on a  $(\text{batch\_size} \cdot \text{seq\_len}) \times \text{hidden\_dim}$  tensor. Here we profile for 16 batches with the hidden dimension fixed to 768 under the standard BERT configuration.

The result tensor needs to be added by the input tensor and normalized after projection and feed forward network of BERT transformer. Rather than launching two separated kernels, we fuse them into a single kernel and re-use data at the register level. In addition to kernel fusion, we leverage FP16 SIMD2 to increase the computational throughput of layernorm by assigning more workload to each thread. We normalize the execution time by that of the optimized layernorm and present the results in Figure 9: the improved version with kernel fusion provides us with a 69% improvement on average over the unfused baseline for sequence lengths ranging 128 to 1024.

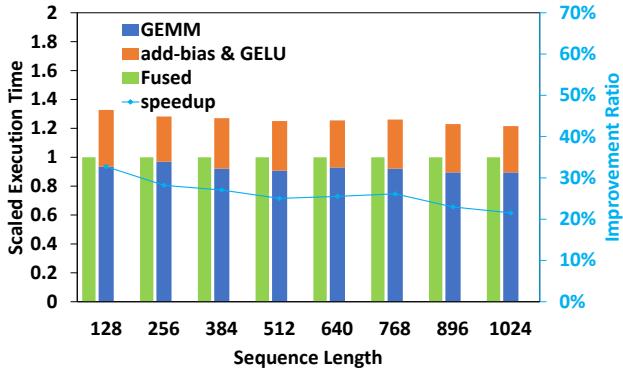


Fig. 10: Kernel fusion for GEMM, add-bias, and GELU. The shape of output tensor is  $(\text{batch\_size} \cdot \text{seq\_len}) \times (\text{scale} \cdot \text{hidden\_dim})$ . Here we profile for 16 batches with the hidden dimension and the scale factor fixed to 768 and 4 under the standard BERT configuration.

### B. Kernel fusion for GEMM and add-bias & activation

Regarding the GEMM, add-bias and activation pattern in BERT transformer, we also provide a fused kernel to reduce the global memory access. An unfused implementation is to call vendor’s GEMM, store the output to global memory, and then load the result matrix from global memory for further element-wise operations. In our optimized version, when the result matrix of GEMM is held in registers, we conduct fused element-wise operations that re-use data at the register level. Once the element-wise transform (add-bias and GELU) is completed, we then store the results to the global memory. Figure 10 compares the performance of fused and unfused versions. In each clustered bar plot, the detailed execution time breakdown of the unfused implementation, normalized by the fused execution time (shown in the left bars), is shown in the stacked bar on the right. By fusing element-wise operations into the GEMM epilogue, we improve the performance by 24% on average for sequence lengths ranging 128 to 1024. It is worth mentioning that we feed *packed* tensors into both fused and non-fused kernels, such that the performance gain in Sec IV A and B are solely from kernel fusion.

### C. Optimizing multi-head attention

Figure 3 shows that MHA accounts for 22% - 49% of the total execution time. We optimize this key algorithm by fusing softmax into GEMMs without calculating for useless padded tokens under variable-length inputs. For short sequences, we hold the intermediate matrix in registers and shared memory. For long sequences, we adopt a grouped GEMM based fused MHA and fuse softmax operations into our customized GEMM epilogue and mainloop to hide the memory latency. In both implementations, the input matrices are accessed according to the position information obtained from the zero padding algorithm so that no redundant calculations are introduced.

Figure 11 compares the MHA performance for sequences shorter than 384. Here cuBLAS denotes the unfused implementation that calls cuBLAS for batched GEMM. The

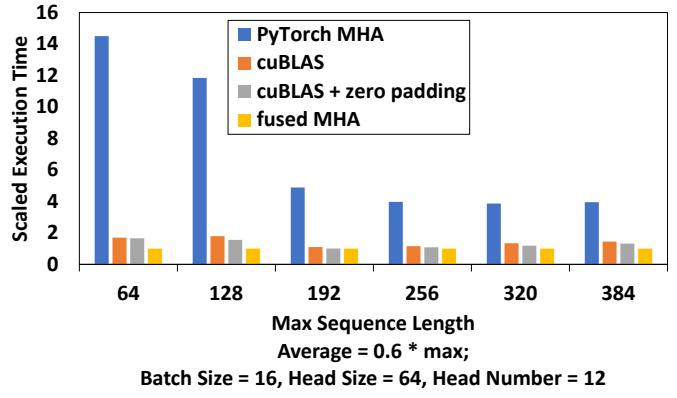


Fig. 11: Fused MHA for short sequences.

softmax operation between two batched GEMM can benefit from the zero padding algorithm, by only accessing unpadded tokens according to the known indices. This variant is denoted as *cuBLAS + zero padding* in the figure. cuBLAS batched GEMM improves the performance over stand PyTorch MHA by 5 folds while enabling the zero padding algorithm for softmax further improves the performance by 9%. Our MHA fully fuses the softmax and two batched GEMMs into one kernel, resulting in average speedups of 617%, 42%, and 30% over all three variants for variable sequence lengths ranging from 64 to 384.

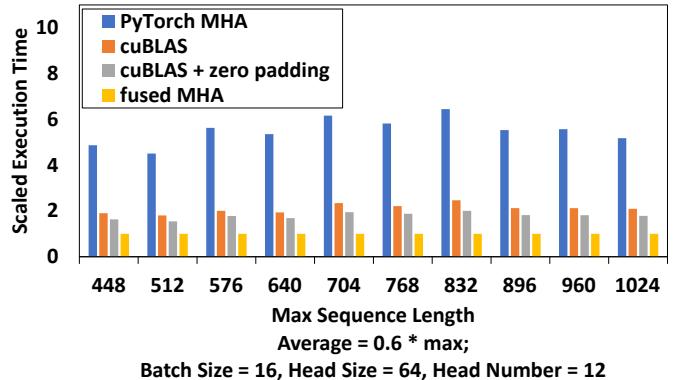


Fig. 12: Fused MHA for long sequences.

Figure 12 compares the performance of the MHA for sequences longer than 448. The cuBLAS batched GEMM triples the MHA performance over PyTorch, while eliminating wasted calculations in softmax further brings a 17% improvement. By introducing the high-performance grouped GEMM and fusing softmax into GEMMs, our fused MHA outperforms the variant MHA implementations by 451%, 110% and 79% for maximal sequence lengths ranging 448 to 1024, where the average sequence length is 60% of the maximum.

Figure 13 compares the scaled execution time of the FMHA module of our ByteTransformer against FlashAttention under the standard BERT setup. As shown in the figure, our FMHA presents advantages for small batch sizes (101% faster on average) while FlashAttention becomes more efficient for

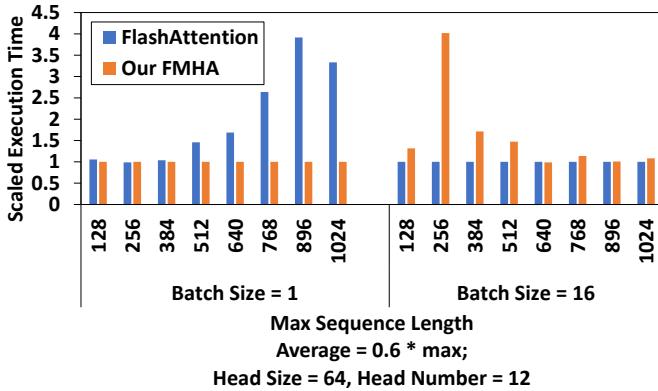


Fig. 13: Comparisons of our FMHA with FlashAttention.

large batch sizes (59% faster on average). This is because FlashAttention maps a whole attention unit to a threadblock, which, although allows for the complete preservation of the intermediate matrix of an attention unit within shared-memory for any sequence length, results in performance degradation when there are insufficient tasks assigned.

#### D. Benchmarking single-layer BERT transformer with step-wise optimizations

Figure 14 compares the performance of a single-layer BERT transformer to reflect our step-wise optimizations. At each step, we add a new optimization upon the previous variant. The baseline transformer implements the workflow in Figure 2 (a) with padding. We then enable kernel fusion for adding bias and layernorm, which corresponds to *layernorm fusion* in the figure. The next step is to fuse adding bias and GELU into GEMM, denoted by *add bias & GELU fusion*. In order to avoid calculating padded tokens for the variable-length inputs, we further propose the zero padding algorithm as shown in Figure 2 (c). This is denoted by *rm padding* in the figure. Our optimized transformer includes our high-performance fused MHA, as well as all previous optimizations.

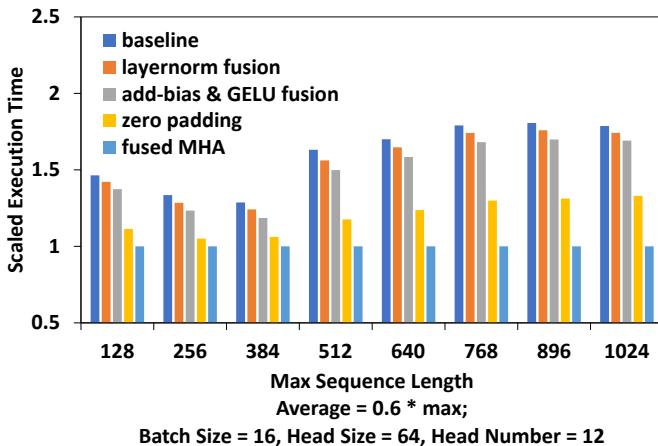


Fig. 14: Single-layer BERT transformer with step-wise optimizations. Each variant includes all previous optimizations.

Fusing adding bias and layernorm into one kernel improves the performance by 3.2%. Fusing adding bias and activation into GEMM epilogue further improves the performance by 3.8%. These two optimizations together improve the overall performance by 7.1%. After bringing in the zero padding algorithm, the redundant calculations are eliminated in most modules other than MHA. We observe a 24% improvement from the previous step. Finally, our fused MHA removes wasted calculations on padded tokens and enables an additional 20% improvement. To summarize, the final version achieves 60% improvement over the baseline version on single-layer BERT.

TABLE III. Single-layer BERT versus E.T. on A100.

Sequence Length	E.T. (ms)	ByteTransformer (ms)	Speedup
256	0.25	0.07	3.57×
1024	1.04	0.09	11.56×

Table III compares the execution time for a single-layer, non-pruned BERT (batch size = 1) between E.T. and ByteTransformer, as E.T. has only open-sourced its single-layer, single-batch prototype. We achieve a speed-up of up to 11 times over E.T., which is optimized specifically for pruned models on legacy Volta GPUs. Since a pruned model can lead to significant reduction in total computations but with possible accuracy trade-offs, we do not include E.T. in our further end-to-end performance evaluations for non-pruned models on an A100 GPU for fairness and comparability.

#### E. Benchmarking end-to-end performance of BERT

The standard BERT transformer is a stacked structure of 12 layers of the encoder module. The output of each encoder module is utilized as an input tensor in the next iteration. Figure 15 shows the end-to-end performance of ByteTransformer and compares it against state-of-the-art transformer implementations: PyTorch with JIT, TensorFlow with XLA acceleration, Microsoft DeepSpeed-Inference, NVIDIA FasterTransformer and Tencent TurboTransformer. We adopt the standard BERT transformer configuration for end-to-end benchmark: 12 heads, head size equal to 64 and 12 iterations (layers). We benchmark for cases whose batch sizes are equal to 1, 8 and 16 and change sequence lengths from 64 to 1024.

Compared with popular DL frameworks PyTorch, TensorFlow, and Microsoft DeepSpeed-Inference, our ByteTransformer achieves 87%, 131%, and 74% faster end-to-end performance on average. When benchmarking Tencent TurboTransformer, we turn on its SmartBatch mode to reach optimal batching performance. Since TurboTransformer only supports sequence lengths smaller than or equal to 512, we do not benchmark longer sequences for it. TurboTransformer regroups and pads similar sequences into a batch so it launches excessive kernels at the run-time. It is faced with significant performance degradation for models with large batch numbers and sequence lengths. NVIDIA FasterTransformer, although it supports long sequences regarding the functionality, its backend TensorRT fused MHA cannot be scaled to long sequences due to the limited register, its end-to-end efficiency cannot

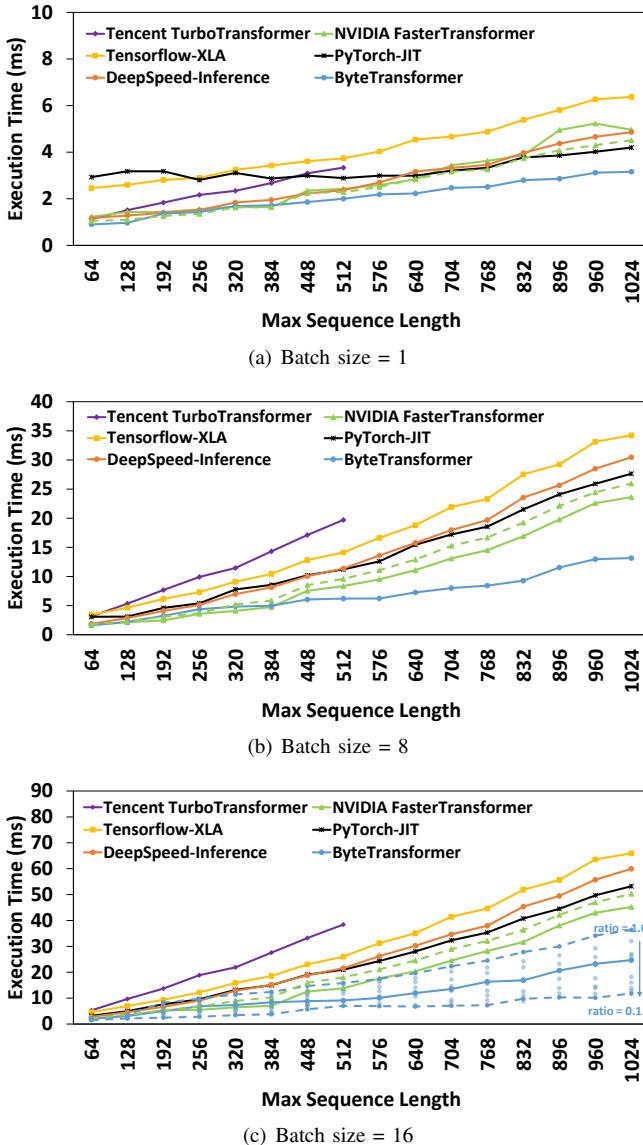


Fig. 15: End-to-end benchmark for standard BERT transformer, head size = 64, head number = 12, layer = 12, average sequence length =  $0.6 * \text{max sequence length}$ .

be maintained when the sequence length becomes longer than 512. Experimental results in Figure 15 show that ByteTransformer outperforms TurboTransformer and FasterTransformer by 138% and 55% on average, respectively.

Figure 15 (c) further includes the end-to-end performance of ByteTransformer for average-to-maximum sequence length ratios ranging from 0.1 to 1.0. The upper dashed blue line represents the execution time of ByteTransformer at a ratio of 1.0, while the lower dashed line corresponds to a ratio of 0.1. Our padding-free algorithm reduces the runtime by up to 66% for a ratio of 0.1 compared to a fixed-sequence-length input. When disabling the support for variable-length inputs of FasterTransformer, as shown by the dashed green lines in Figure 15, we observe a moderate decrease in performance for

larger batch sizes (batch sizes = 8 and 16) but an improvement in performance for a small batch size (batch size = 1). In contrast, our FMHA-enabled padding-free algorithm significantly improves the performance of the end-to-end BERT transformer for variable-length input with an average-to-maximum ratio of 0.6, outpacing NVIDIA FasterTransformer by a notable difference of 54% to 16%.

TABLE IV. Configurations of other BERT-like transformers.

Model	layer number	head number	head size
ALBERT	12	16	64
DistilBERT	6	12	64
DeBERTa	12	12	64

#### F. Extending to other BERT-like transformers

We extend the optimizations on kernel fusion and the padding-free algorithm presented in our work to other BERT-like transformers, including ALBERT, DistilBERT, and DeBERTa. Table IV summarizes the model configurations, and readers can refer to [34]–[36] for more detailed information about their architectures. Figure 16 compares the performance of the ByteTransformer with state-of-the-art DL frameworks under these models. Following the setup for our demonstrated standard BERT benchmarks, the average sequence length is set to 60% of the maximal sequence length. TurboTransformer only supports sequences shorter than 512, so its performance data for long sequences are not presented. FasterTransformer and TurboTransformer do not support DeBERTa, so their results are not included in that model. It is worth noting that TensorFlow encountered an out-of-memory error for sequence length 1024 in the DeBERTa model, resulting in this data point being excluded. For ALBERT and DistilBERT, our ByteTransformer on average outperforms PyTorch, TensorFlow, Tencent TurboTransformer, DeepSpeed-Inference, and NVIDIA FasterTransformer by 98%, 158%, 256%, 93%, and 53%, respectively. For the DeBERTa model, our ByteTransformer outperforms PyTorch, TensorFlow, and DeepSpeed by 44%, 243%, and 74%, respectively.

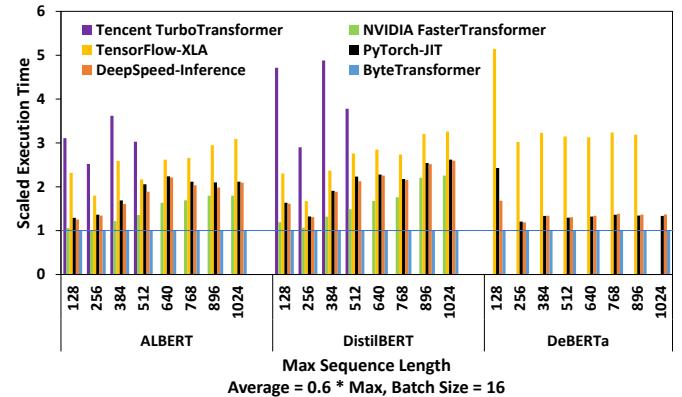


Fig. 16: End-to-end benchmark for other BERT-like models.

## V. CONCLUSIONS

We have presented ByteTransformer, a high-performance transformer optimized for variable-length sequences. ByteTransformer not only brings algorithmic level innovation that frees the transformer from padding overhead, but also incorporates architecture-aware optimizations to accelerate functioning modules of the transformer. Our optimized fused MHA, as well as other step-wise optimizations, together provide us with significant speedup over current state-of-the-art transformers. The end-to-end performance of the standard BERT transformer benchmarked on an NVIDIA A100 GPU demonstrates that our ByteTransformer surpasses PyTorch, TensorFlow, Tencent TurboTransformer, Microsoft DeepSpeed-Inference, and NVIDIA FasterTransformer by 87%, 131%, 138%, 74% and 55%, respectively. Moreover, we have shown that our optimizations are not specific to BERT, but can be applied to other BERT-like transformers, including ALBERT, DistilBERT, and DeBERTa. We are striving to make ByteTransformer completely open-source. This will allow the wider research community to benefit from our optimized implementation and to continue advancing the field. We are also dedicated to further expanding the presented strategies to accelerate a wider range of BERT-like transformer models, both in inference and training.

## REFERENCES

- [1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.
- [3] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, “Xlnet: Generalized autoregressive pretraining for language understanding,” *Advances in neural information processing systems*, vol. 32, 2019.
- [4] S. Edunov, M. Ott, M. Auli, and D. Grangier, “Understanding back-translation at scale,” *arXiv preprint arXiv:1808.09381*, 2018.
- [5] Q. Chen, H. Zhao, W. Li, P. Huang, and W. Ou, “Behavior sequence transformer for e-commerce recommendation in alibaba,” in *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, 2019, pp. 1–4.
- [6] F. Sun, J. Liu, J. Wu, C. Pei, X. Lin, W. Ou, and P. Jiang, “Bert4rec: Sequential recommendation with bidirectional encoder representations from transformer,” in *Proceedings of the 28th ACM international conference on information and knowledge management*, 2019, pp. 1441–1450.
- [7] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever et al., “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [8] J. Zeng, M. Li, Z. Wu, J. Liu, Y. Liu, D. Yu, and Y. Ma, “Boosting distributed training performance of the unpadded bert model,” *arXiv preprint arXiv:2208.08124*, 2022.
- [9] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell et al., “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [10] J. Fang, Y. Yu, C. Zhao, and J. Zhou, “Turbotransformers: an efficient gpu serving system for transformer models,” in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 389–402.
- [11] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, “Zero: Memory optimizations toward training trillion parameter models,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.
- [12] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, “Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters,” in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3505–3506.
- [13] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, “Megatron-lm: Training multi-billion parameter language models using model parallelism,” *arXiv preprint arXiv:1909.08053*, 2019.
- [14] X. Wang, Y. Xiong, X. Qian, Y. Wei, L. Li, and M. Wang, “Lightseq2: Accelerated training for transformer-based models on gpus,” *arXiv preprint arXiv:2110.05722*, 2021.
- [15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., “{TensorFlow}: a system for {Large-Scale} machine learning,” in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.
- [16] Google, <https://www.tensorflow.org/xla>, Retrieved in 2022, online.
- [17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.
- [18] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, “On the properties of neural machine translation: Encoder-decoder approaches,” *arXiv preprint arXiv:1409.1259*, 2014.
- [19] NVIDIA , <https://developer.nvidia.com/tensorrt>, Retrieved in 2022, online.
- [20] Y. Zhai, E. Giem, Q. Fan, K. Zhao, J. Liu, and Z. Chen, “Ft-blas: a high performance blas implementation with online fault tolerance,” in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 127–138.
- [21] K. Zhao, S. Di, S. Li, X. Liang, Y. Zhai, J. Chen, K. Ouyang, F. Cappello, and Z. Chen, “Algorithm-based fault tolerance for convolutional neural networks,” *IEEE Transactions on Parallel and Distributed Systems*, 2020.
- [22] Y. Zhai, M. Ibrahim, Y. Qiu, F. Boemer, Z. Chen, A. Titov, and A. Lyshevsky, “Accelerating encrypted computing on intel gpus,” in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 705–716.
- [23] NVIDIA , <https://github.com/NVIDIA/FasterTransformer>, Retrieved in 2022, online.
- [24] NVIDIA , <https://developer.nvidia.com/cublas>, Retrieved in 2022, online.
- [25] S. Chen, S. Huang, S. Pandey, B. Li, G. R. Gao, L. Zheng, C. Ding, and H. Liu, “Et: re-thinking self-attention for transformer models on gpus,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–18.
- [26] R. Y. Aminabadi, S. Rajbhandari, M. Zhang, A. A. Awan, C. Li, D. Li, E. Zheng, J. Rasley, S. Smith, O. Ruwase et al., “Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale,” *arXiv preprint arXiv:2207.00032*, 2022.
- [27] PyTorch, <https://pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.html>, Retrieved in 2022, online.
- [28] NVIDIA , <https://github.com/NVIDIA/TensorRT/tree/main/plugin/bertQKVToContextPlugin>, Retrieved in 2022, online.
- [29] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, “Flashattention: Fast and memory-efficient exact attention with io-awareness,” *arXiv preprint arXiv:2205.14135*, 2022.
- [30] D. Hendrycks and K. Gimpel, “Gaussian error linear units (gelus),” *arXiv preprint arXiv:1606.08415*, 2016.
- [31] NVIDIA , <https://github.com/NVIDIA/cutlass>, Retrieved in 2022, online.
- [32] NVIDIA , [https://github.com/NVIDIA/cutlass/tree/master/examples/41\\_multi\\_head\\_attention](https://github.com/NVIDIA/cutlass/tree/master/examples/41_multi_head_attention), Retrieved in 2022, online.
- [33] NVIDIA , [https://github.com/NVIDIA/cutlass/blob/master/include/cutlass/gemm/kernel/grouped\\_problem\\_visitor.h#L203-L322](https://github.com/NVIDIA/cutlass/blob/master/include/cutlass/gemm/kernel/grouped_problem_visitor.h#L203-L322), Retrieved in 2022, online.
- [34] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, “Albert: A lite bert for self-supervised learning of language representations,” *arXiv preprint arXiv:1909.11942*, 2019.
- [35] V. Sanh, L. Début, J. Chaumond, and T. Wolf, “Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter,” *arXiv preprint arXiv:1910.01108*, 2019.
- [36] P. He, X. Liu, J. Gao, and W. Chen, “Deberta: Decoding-enhanced bert with disentangled attention,” *arXiv preprint arXiv:2006.03654*, 2020.



# ORCA: A Distributed Serving System for Transformer-Based Generative Models

Gyeong-In Yu and Joo Seong Jeong, *Seoul National University*;  
Geon-Woo Kim, *FriendliAI and Seoul National University*; Soojeong Kim, *FriendliAI*;  
Byung-Gon Chun, *FriendliAI and Seoul National University*

<https://www.usenix.org/conference/osdi22/presentation/yu>

This paper is included in the Proceedings of the  
16th USENIX Symposium on Operating Systems  
Design and Implementation.

July 11-13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the  
16th USENIX Symposium on Operating  
Systems Design and Implementation  
is sponsored by

NetApp®

# ORCA: A Distributed Serving System for Transformer-Based Generative Models

Gyeong-In Yu  
*Seoul National University*

Joo Seong Jeong  
*Seoul National University*

Geon-Woo Kim  
*FriendliAI*  
*Seoul National University*

Soojeong Kim  
*FriendliAI*

Byung-Gon Chun<sup>\*</sup>  
*FriendliAI*  
*Seoul National University*

## Abstract

Large-scale Transformer-based models trained for generation tasks (e.g., GPT-3) have recently attracted huge interest, emphasizing the need for system support for serving models in this family. Since these models generate a next token in an autoregressive manner, one has to run the model multiple times to process an inference request where each iteration of the model generates a single output token for the request. However, existing systems for inference serving do not perform well on this type of workload that has a multi-iteration characteristic, due to their inflexible scheduling mechanism that cannot change the current batch of requests being processed; requests that have finished earlier than other requests in a batch cannot return to the client, while newly arrived requests have to wait until the current batch completely finishes.

In this paper, we propose iteration-level scheduling, a new scheduling mechanism that schedules execution at the granularity of iteration (instead of request) where the scheduler invokes the execution engine to run only a single iteration of the model on the batch. In addition, to apply batching and iteration-level scheduling to a Transformer model at the same time, we suggest selective batching, which applies batching only to a selected set of operations. Based on these two techniques, we have implemented a distributed serving system called ORCA, with additional designs for scalability to models with hundreds of billions of parameters. Our evaluation on a GPT-3 175B model shows that ORCA can significantly outperform NVIDIA FasterTransformer in terms of both latency and throughput: 36.9 $\times$  throughput improvement at the same level of latency.

## 1 Introduction

Language generation tasks are becoming increasingly paramount to many types of applications, such as chatbot [9, 52], summarization [41, 45, 54], code generation [13], and caption generation [65, 66]. Moreover, recent works published by

AI21 Labs [37], DeepMind [26, 48], Google [15, 21, 63], Meta Platforms [10, 67], Microsoft [50], Microsoft & NVIDIA [59], and OpenAI [12] have reported that every language processing task, including translation [11, 17], classification [20, 53], question-answering [32, 33, 40] and more, can be cast as a language generation problem and have shown great improvements along this direction. The rise of generative models is not limited to the language domain; the AI community has also given growing interest to generation problems in other domains such as image, video, speech, or a mixture of multiple domains [19, 38, 51, 62]. At the heart of generative models lies the Transformer architecture [60] and its variants [15, 47–49]. By relying on the attention mechanism [60], Transformer models can learn better representations where each element of the sequence may have a direct connection with every other element, which was not possible in recurrent models [25].

To use generative models in real-world applications, we often delegate the inference procedure to a separate service responsible for ML inference serving. The growing demands for this service, which should provide inference results for client requests at low latency and high throughput, have facilitated the development of inference serving systems such as Triton Inference Server [7] and TensorFlow Serving [42]. These systems can use a separately-developed DNN *execution engine* to perform the actual tensor operations. For example, we can deploy a service for language generation tasks by using a combination of Triton and FasterTransformer [4], an execution engine optimized for the inference of Transformer-based models. In this case, Triton is mainly responsible for grouping multiple client requests into a batch, while FasterTransformer receives the batch from Triton and conducts the inference procedure in the batched manner.

Unfortunately, we notice that the existing inference systems, including both the serving system layer and the execution engine layer, have limitations in handling requests for Transformer-based generative models. Since these models are trained to generate a next token in an autoregressive manner, one should run the model as many times as the number of tokens to generate, while for other models like ResNet [24] and

<sup>\*</sup>Corresponding author.

BERT [18] a request can be processed by running the model once. That is, in order to process a request to the generative model, we have to run multiple *iterations* of the model; each iteration generates a single output token, which is used as an input in the following iteration. Such multi-iteration characteristic calls into question the current design of inference systems, where the serving system schedules the execution of the engine at the granularity of request. Under this design, when the serving system dispatches a batch of requests to the engine, the engine returns inference results for the entire batch at once after processing all requests within the batch. As different client requests may require different numbers of iterations for processing, requests that have finished earlier than others in the batch cannot return to the client, resulting in an increased latency. Requests arrived after dispatching the batch also should wait for processing the batch, which can significantly increase the requests’ queueing time.

In this paper, we propose to schedule the execution of the engine *at the granularity of iteration* instead of request. In particular, the serving system invokes the engine to run only a single iteration of the model on the batch. As a result, a newly arrived request can be considered for processing after waiting for only a single iteration of the model. The serving system checks whether a request has finished processing after every return from the engine – hence the finished requests can also be returned to the clients immediately.

Nevertheless, a noticeable challenge arises when we attempt to apply batching and the iteration-level scheduling at the same time. Unlike the canonical request-level scheduling, the proposed scheduling can issue a batch of requests where each request has so far processed a different number of tokens. In such a case, the requests to the Transformer model cannot be processed in the batched manner because the attention mechanism calls for non-batchable tensor operations whose input tensors have variable shapes depending on the number of processed tokens.

To address this challenge, we suggest to apply batching only to a selected set of operations, which we call *selective batching*. By taking different characteristics of operations into account, selective batching splits the batch and processes each request individually for the Attention<sup>1</sup> operation while applying batching to other operations of the Transformer model. We observe that the decision not to batch the executions of the Attention operation has only a small impact on efficiency. Since the Attention operation is not associated with any model parameters, applying batching to Attention has no benefit of reducing the amount of GPU memory reads by reusing the loaded parameters across multiple requests.

Based on these techniques, we design and implement ORCA, a distributed serving system for Transformer-based generative models. In order to handle large-scale models,

<sup>1</sup>In some literature the Attention operation has an extended definition that includes linear layers (QKV Linear and Attn Out Linear; Figure 1b). On the other hand, we use a narrow definition as described in Figure 1b.

ORCA adopts parallelization strategies including intra-layer and inter-layer model parallelism, which were originally developed by training systems [55, 58] for Transformer models. We also devise a new scheduling algorithm for the proposed iteration-level scheduling, with additional considerations for memory management and pipelined execution across workers.

We evaluate ORCA using OpenAI GPT-3 [12] models with various configurations, scaling up to 341B of parameters. The results show that ORCA significantly outperforms FasterTransformer [4], showing  $36.9 \times$  throughput improvement at the same level of latency. While we use a language model as a driving example throughout the paper and conduct experiments only on language models, generative models in other domains can benefit from our approach as long as the models are based on the Transformer architecture and use the autoregressive generation procedure [19, 38, 51, 62].

## 2 Background

We provide background on the inference procedure of GPT [12, 47], a representative example of Transformer-based generative models that we use throughout this paper, and ML inference serving systems.

**Inference procedure of GPT.** GPT is an autoregressive language model based on one of architectural variants of Transformer [60]. It takes text as input and produces new text as output. In particular, the model receives a sequence of input tokens and then completes the sequence by generating subsequent output tokens. Figure 1a illustrates a simplified computation graph that represents this procedure with a three-layer GPT model, where nodes and edges indicate Transformer layers and dependencies between the layers, respectively. The Transformer layers are executed in the order denoted by the numbers on the nodes, and the nodes that use the same set of model parameters (i.e., nodes representing the same layer) are filled with the same color.

The generated output token is fed back into the model to generate the next output token, imposing a sequential, one-by-one inference procedure. This autoregressive procedure of generating a single token is done by running all the layers of the model with the input, which is either a sequence of input tokens that came from the client or a previously generated output token. We define the run of all layers as an *iteration* of the model. In the example shown in Figure 1a, the inference procedure comprises three iterations. The first iteration (“iter 1”) takes all the input tokens (“I think this”) at once and generates the next token (“is”). This iteration composes an *initiation phase*, a procedure responsible for processing the input tokens and generating the first output token. The next two iterations (“iter 2” and “iter 3”), which compose an *increment phase*, take the output token of the preceding iteration and generate

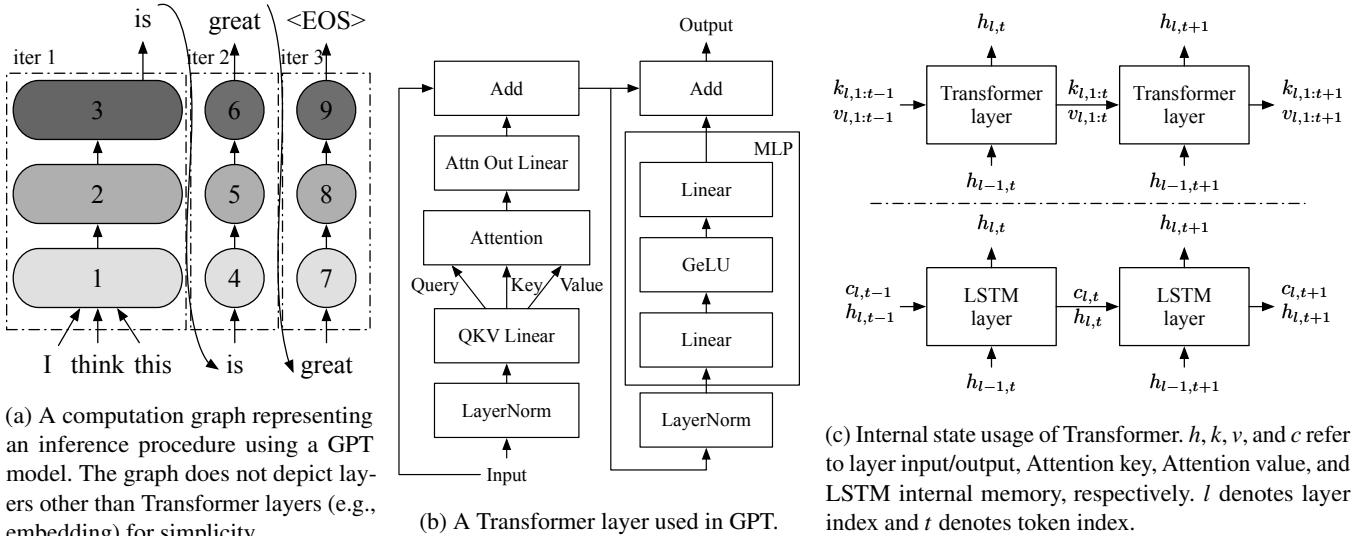


Figure 1: Illustrations for GPT’s inference procedure, Transformer layer, and internal state usage.

the next token. In this case, “iter 3” is the last iteration because it produces “<EOS>”, a special end-of-sequence token that terminates output generation. Note that while the increment phase comprises multiple iterations because each iteration is only able to process a single token, the initiation phase is typically implemented as a single iteration by processing all the input tokens in parallel.

The original Transformer [60] employs two stacks of Transformer layers, while GPT’s architecture consists of a single layer stack, namely decoder. Figure 1b shows a Transformer layer used in GPT. Among the operations that compose the Transformer layer, *Attention* is the essence that distinguishes Transformer from other architectures. At a high level, the Attention operation computes a weighted average of the tokens of interest so that each token in the sequence is aware of the other. It takes three inputs, query, key, and value, computes dot products of the query (for the current token) with all keys (for the tokens of interest), applies Softmax on the dot products to get weights, and conducts weighted average of all values associated with the weights.

Since the Attention requires keys and values of all preceding tokens,<sup>2</sup> we consider the keys and values as internal states that should be maintained across multiple iterations. A naïve, state-less inference procedure would take all tokens in the sequence (including both the client-provided input tokens and the output tokens generated so far) to recompute all the keys and values at every iteration. To avoid such recomputation, fairseq [43] suggests incremental decoding, which saves the keys and values for reuse in successive iterations. Other systems for Transformer such as FasterTransformer [4] and Megatron-LM [3] also do the same.

<sup>2</sup>Language models like GPT use causal masking, which means all preceding tokens are of interest and participate in the Attention operation.

Figure 1c illustrates the state usage pattern of Transformer, along with LSTM [25] that also maintains internal states. The main difference is that the size of the states ( $k$  for Attention key and  $v$  for value) in Transformer increases with iteration, whereas the size of the states ( $c$  for LSTM internal memory and  $h$  for LSTM layer’s input/output) in LSTM remains constant. When processing the token at index  $t$ , the Attention operation takes all previous Attention keys  $k_{l,1:t-1}$  and values  $v_{l,1:t-1}$  along with the current key  $k_{l,t}$  and value  $v_{l,t}$ .<sup>3</sup> Therefore, the Attention operation should perform computation on tensors of different shapes depending on the number of tokens already processed.

Prior to the Attention operation, there are the layer normalization operation (LayerNorm) and the QKV Linear (linear and split operations to get the query, key and value). Operations performed after Attention are, in order, a linear operation (Attn Out Linear), an add operation for residual connection (Add), layer normalization operation (LayerNorm), the multi-layer perceptron (MLP) operations, and the other residual connection operation (Add).

**ML inference serving systems.** Growing demands for ML-driven applications have made ML inference serving service a critical workload in modern datacenters. Users (either the end-user or internal microservices of the application) submit requests to an inference service, and the service gives replies on the requests based on a pre-defined ML model using its provisioned resource, typically equipped with specialized accelerators such as GPUs and TPUs. In particular, the service runs a DNN model with input data to generate output for the

<sup>3</sup> $k_{l,1:t-1}$  represents Attention keys of the  $l$ -th layer for tokens at indices 1 to  $t-1$  while  $k_{l,t}$  is for the Attention key of the  $l$ -th layer for the token at index  $t$ . Same for  $v_{l,1:t-1}$  and  $v_{l,t}$ .

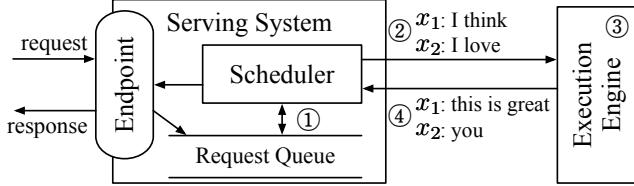


Figure 2: Overall workflow of serving a generative language model with existing systems.

request. Just like other services operating on datacenters, a well-managed inference service should provide low latency and high throughput within a reasonable amount of cost.

To meet such constraints, service operators often use ML inference serving systems such as Triton Inference Server [7] and TensorFlow Serving [42]. These systems can be seen as an abstraction sitting atop underlying model *execution engines* such as TensorRT [6], TVM [14], TensorFlow [8], and many others [44, 46], being agnostic to various kinds of ML models, execution engines, and computing hardware. While delegating the role of driving the main mathematical operations to the engines, serving systems are in charge of exposing endpoints that receive inference requests, scheduling executions of the engine, and sending responses to the requests. Accordingly, these systems focus on aspects such as batching the executions [7, 16, 35, 42, 56], selecting an appropriate model from multiple model variants [16, 27, 30, 57], deploying multiple models (each for different inference services) on the same device [7, 29, 35, 56], and so on.

Among the features and optimizations provided by serving systems, batching is a key to achieve high accelerator utilization when using accelerators like GPUs. When we run the execution engine with batching enabled, the input tensors from multiple requests coalesce into a single, large input tensor before being fed to the first operation of the model. Since the accelerators prefer large input tensors over small ones to better exploit the vast amount of parallel computation units, the engine’s throughput is highly dependent on the batch size, i.e., the number of inference requests the engine processes together. Reusing the model parameters loaded from off-chip memory is another merit in batched execution, especially when the model involves memory-intensive operations.

Figure 2 shows an overall workflow of serving a generative language model with existing serving systems and execution engines. The main component of the serving system (e.g., Triton [7]) is the scheduler, which is responsible for ① creating a batch of requests by retrieving requests from a queue and ② scheduling the execution engine (e.g., FasterTransformer [4]) to process the batch. The execution engine ③ processes the received batch by running multiple iterations of the model being served and ④ returns the generated text back to the serving system. In Figure 2, the serving system schedules the engine to process two requests ( $x_1$ : “I think”,  $x_2$ : “I love”) in

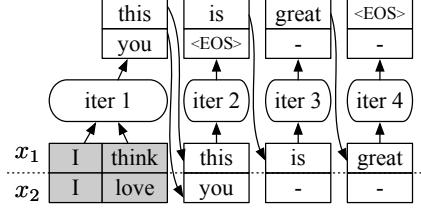


Figure 3: An illustration for a case where the requests have the same input length but some requests finish earlier than others. Shaded tokens represent input tokens. “-” denotes inputs and outputs of extra computation imposed by the scheduling.

a batch and the engine generates “this is great” and “you” for requests  $x_1$  and  $x_2$ , respectively.

### 3 Challenges and Proposed Solutions

In this section, we describe challenges in serving Transformer-based generative models and propose two techniques: iteration-level scheduling and selective batching.

**C1: Early-finished and late-joining requests.** One major limitation of existing systems is that the serving system and the execution engine interact with each other only when (1) the serving system schedules the next batch on an idle engine; or (2) the engine finishes processing the current batch. In other words, these systems are designed to schedule executions at *request* granularity; the engine maintains a batch of requests fixed until all requests in the batch finish. This can be problematic in the serving of generative models, since each request in a batch may require different number of iterations, resulting in certain requests finishing earlier than the others. In the example shown in Figure 3, although request  $x_2$  finishes earlier than request  $x_1$ , the engine performs computation for both “active” and “inactive” requests throughout all iterations. Such extra computation for inactive requests ( $x_2$  at iter 3 and 4) limits the efficiency of batched execution.

What makes it even worse is that this behavior prevents an early return of the finished request to the client, imposing a substantial amount of extra latency. This is because the engine only returns the execution results to the serving system when it finishes processing all requests in the batch. Similarly, when a new request arrives in the middle of the current batch’s execution, the aforementioned scheduling mechanism makes the newly arrived request wait until all requests in the current batch have finished. We argue that the current request-level scheduling mechanism cannot efficiently handle workloads with multi-iteration characteristic. Note that this problem of early-finished and late-joining requests does not occur in the training of language models; the training procedure finishes processing the whole batch in a single iteration by using the teacher forcing technique [64].

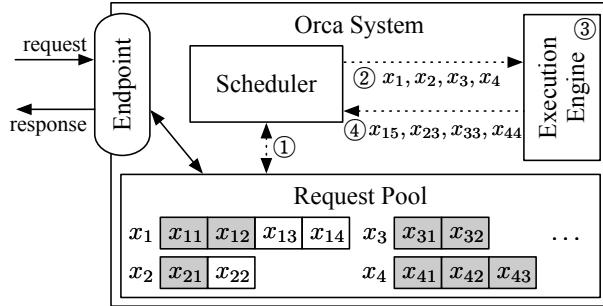


Figure 4: System overview of ORCA. Interactions between components represented as dotted lines indicate that the interaction takes place at every iteration of the execution engine.  $x_{ij}$  is the  $j$ -th token of the  $i$ -th request. Shaded tokens represent input tokens received from the clients, while unshaded tokens are generated by ORCA. For example, request  $x_1$  initially arrived with two input tokens ( $x_{11}, x_{12}$ ) and have run two iterations so far, where the first and second iterations generated  $x_{13}$  and  $x_{14}$ , respectively. On the other hand, request  $x_3$  only contains input tokens ( $x_{31}, x_{32}$ ) because it has not run any iterations yet.

**S1: Iteration-level scheduling.** To address the above limitations, we propose to schedule executions at the granularity of *iteration*. At high level, the scheduler repeats the following procedure: (1) selects requests to run next; (2) invokes the engine to execute *one iteration* for the selected requests; and (3) receives execution results for the scheduled iteration. Since the scheduler receives a return on every iteration, it can detect the completion of a request and immediately return its generated tokens to the client. For a newly arrived request, the request gets a chance to start processing (i.e., the scheduler may select the new request to run next) after execution of the currently scheduled iteration, significantly reducing the queueing delay. With iteration-level scheduling, the scheduler has a full control on how many and which requests are processed in each iteration.

Figure 4 depicts the system architecture and the overall workflow of ORCA using the iteration-level scheduling. ORCA exposes an *endpoint* (e.g., HTTPS or gRPC) where inference requests arrive at the system and responses to the requests are sent out. The endpoint puts newly arrived requests in the *request pool*, a component that manages all requests in the system during their lifetime. The pool is monitored by the *scheduler*, which is responsible for: selecting a set of requests from the pool, scheduling the *execution engine* to run an iteration of the model on the set, receiving execution results (i.e., output tokens) from the engine, and updating the pool by appending each output token to the corresponding request. The engine is an abstraction for executing the actual tensor operations, which can be parallelized across multiple GPUs spread across multiple machines. In the example shown in Figure 4, the scheduler ① interacts with the request pool to

decide which requests to run next and ② invokes the engine to run four selected requests:  $(x_1, x_2, x_3, x_4)$ . The scheduler provides the engine with input tokens of the requests scheduled for the first time. In this case,  $x_3$  and  $x_4$  have not run any iterations yet, so the scheduler hands over  $(x_{31}, x_{32})$  for  $x_3$  and  $(x_{41}, x_{42}, x_{43})$  for  $x_4$ . The engine ③ runs an iteration of the model on the four requests and ④ returns generated output tokens  $(x_{15}, x_{23}, x_{33}, x_{44})$ , one for each scheduled request. Once a request has finished processing, the request pool removes the finished request and notifies the endpoint to send a response. Unlike the method shown in Figure 2 that should run multiple iterations on a scheduled batch until finish of all requests within the batch, ORCA’s scheduler can change which requests are going to be processed at every iteration. We describe the detailed algorithm about how to select the requests at every iteration in Section 4.2.

**C2: Batching an arbitrary set of requests.** When we try to use the iteration-level scheduling in practice, one major challenge that we are going to face is batching. To achieve high efficiency, the execution engine should be able to process any selected set of requests in the batched manner. Without batching, one would have to process each selected request one by one, losing out on the massively parallel computation capabilities of GPUs.

Unfortunately, there is no guarantee that even for a pair of requests  $(x_i, x_j)$ , for the next iteration, their executions can be merged and replaced with a batched version. There are three cases for a pair of requests where the next iteration cannot be batched together: (1) both requests are in the initiation phase and each has different number of input tokens (e.g.,  $x_3$  and  $x_4$  in Figure 4); (2) both are in the increment phase and each is processing a token at different index from each other ( $x_1$  and  $x_2$ ); or (3) each request is in the different phase: initiation or increment ( $x_1$  and  $x_3$ ). Recall that in order to batch the execution of multiple requests, the execution of each request must consist of identical operations, each consuming identically-shaped input tensors. In the first case, the two requests cannot be processed in a batch because the “length” dimension of their input tensors, which is the number of input tokens, are not equal. The requests in the second case have difference in the tensor shape of Attention keys and values because each processes token at different index, as shown in Figure 1c. For the third case, we cannot batch the iterations of different phases because they take different number of tokens as input; an iteration of the initiation phase processes all input tokens in parallel for efficiency, while in the increment phase each iteration takes a single token as its input (we assume the use of fairseq-style incremental decoding [43]).

Batching is only applicable when the two selected requests are in the same phase, with the same number of input tokens (in case of the initiation phase) or with the same token index (in case of the increment phase). This restriction significantly reduces the likelihood of batching in real-world workloads,

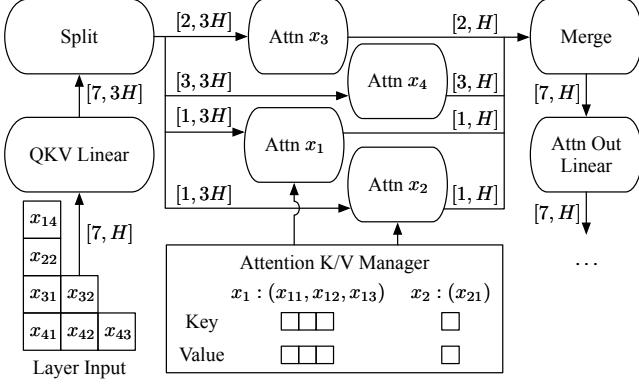


Figure 5: An illustration of ORCA execution engine running a Transformer layer on a batch of requests with selective batching. We only depict the QKV Linear, Attention, and Attention Out Linear operations for simplicity.

because the scheduler should *make a wish* for the presence of two requests eligible for batching at the same time. The likelihood further decreases exponentially as the batch size increases, making it impractical to use a large batch size that can pull out better throughput without compromising latency.

**S2: Selective batching.** We propose *selective batching*, a technique for batched execution that allows high flexibility in composing requests as a batch. Instead of processing a batch of requests by “batchifying” all tensor operations composing the model, this technique selectively apply batching only to a handful of operations.

The main problem regarding batching described above is that the three aforementioned cases<sup>4</sup> correspond to irregularly shaped input (or state) tensors, which cannot be coalesced into a single large tensor and fed into a batch operation. In the canonical batching mechanism, at each iteration, a Transformer layer takes a 3-dimensional input tensor of shape  $[B, L, H]$  generated by concatenating multiple  $[L, H]$  input tensors of requests in a batch, where  $B$  is the batch size,  $L$  is the number of tokens processed together, and  $H$  is the hidden size of the model. For example, in Figure 3, “iter 1” (initiation phase) takes an input tensor of shape  $[2, 2, H]$  and “iter 2” (increment phase) takes a tensor of shape  $[2, 1, H]$ . However, when the scheduler decides to run an iteration on batch  $(x_1, x_2, x_3, x_4)$  in Figure 4, the inputs for requests in the initiation phase ( $x_3 : [2, H]$  and  $x_4 : [3, H]$ ) cannot coalesce into a single tensor of shape  $[B, L, H]$  because  $x_3$  and  $x_4$  have different number of input tokens, 2 and 3.

Interestingly, not all operations are incompatible with such irregularly shaped tensors. Operations such as non-Attention matrix multiplication and layer normalization can be made to work with irregularly shaped tensors by flattening the tensors.

<sup>4</sup>We use the first case as a driving example, but the argument can be similarly applied to the other two cases.

For instance, the aforementioned input tensors for  $x_3$  and  $x_4$  can compose a 2-dimensional tensor of shape  $[\sum L, H] = [5, H]$  without an explicit batch dimension. This tensor can be fed into all non-Attention operations including Linear, Layer-Norm, Add, and GeLU operations because they do not need to distinguish tensor elements of different requests. On the other hand, the Attention operation requires a notion of requests (i.e., requires the batch dimension) to compute attention only between the tokens of the same request, typically done by applying cuBLAS routines for batch matrix multiplication.

Selective batching is aware of the different characteristics of each operation; it splits the batch and processes each request individually for the Attention operation while applying token-wise (instead of request-wise) batching to other operations without the notion of requests. Figure 5 presents the selective batching mechanism processing a batch of requests  $(x_1, x_2, x_3, x_4)$  described in Figure 4. This batch has 7 input tokens to process, so we make the input tensor have a shape of  $[7, H]$  and apply the non-Attention operations. Before the Attention operation, we insert a Split operation and run the Attention operation separately on the split tensor for each request. The outputs of Attention operations are merged back into a tensor of shape  $[7, H]$  by a Merge operation, bringing back the batching functionality to the rest of operations.

To make the requests in the increment phase can use the Attention keys and values for the tokens processed in previous iterations, ORCA maintains the generated keys and values in the *Attention K/V manager*. The manager maintains these keys and values separately for each request until the scheduler explicitly asks to remove certain request’s keys and values, i.e., when the request has finished processing. The Attention operation for request in the increment phase ( $x_1$  and  $x_2$ ) takes keys and values of previous tokens ( $x_{11}, x_{12}, x_{13}$  for  $x_1$ ;  $x_{21}$  for  $x_2$ ) from the manager, along with the current token’s query, key, and value from the Split operation to compute attention between the current token and the previous ones.

## 4 ORCA Design

Based on the above techniques, we design and implement ORCA: a distributed serving system for Transformer-based generative models. We have already discussed the system components and the overall execution model of ORCA while describing Figure 4. In this section, we answer the remaining issues about how to build an efficient system that can scale to large-scale models with hundreds of billions of parameters. We also describe the scheduling algorithm for iteration-level scheduling, i.e., how to select a batch of requests from the request pool at every iteration.

### 4.1 Distributed Architecture

Recent works [12, 31] have shown that scaling language models can dramatically improve the quality of models. Hence,

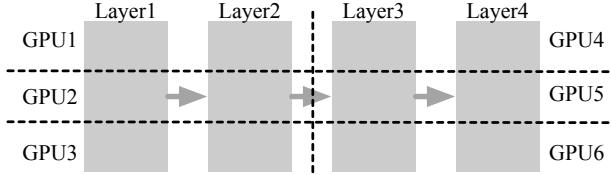


Figure 6: An example of intra- and inter- layer parallelism. A vertical dotted line indicates partitioning between layers and a horizontal line indicates partitioning within a layer.

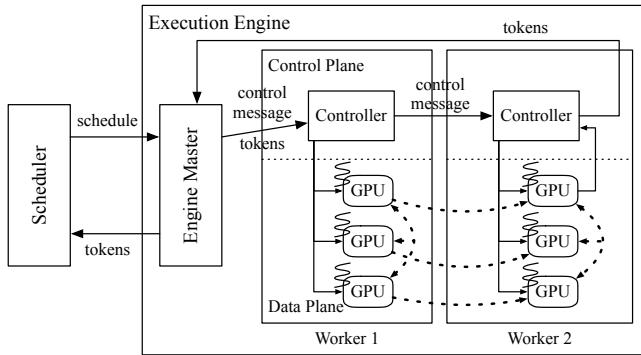


Figure 7: An illustration of the distributed architecture of ORCA’s execution engine using the parallelization configuration shown in Figure 6. For example, the first inter-layer partition (Layer1 and Layer2) in Figure 6 is assigned to Worker1, while the second partition is assigned to Worker2.

system support for serving such large language models is getting more importance, especially when the model does not fit in a single GPU. In such a case, one should split the model parameters along with the corresponding computation and distribute them across multiple GPUs and machines.

ORCA composes known parallelization techniques for Transformer models: intra-layer parallelism and inter-layer parallelism. These two model parallelism strategies, which are also used by FasterTransformer [4], have been originally developed for distributed training. Intra-layer parallelism [55, 58] splits matrix multiplications (i.e., Linear and Attention operations) and their associated parameters over multiple GPUs. We omit the detail about how this strategy partitions each matrix multiplication. On the other hand, inter-layer parallelism splits Transformer layers over multiple GPUs. ORCA assigns the same number of Transformer layers to each GPU. Figure 6 illustrates an example application of intra- and inter- layer parallelism to a 4-layer GPT model. The 4 layers are split into 2 inter-layer partitions, and the layers in the partition are subdivided into 3 intra-layer partitions. We assign each partition to a GPU, using a total of 6 GPUs.

The ORCA execution engine supports distributed execution using the techniques described above. Figure 7 depicts the architecture of an ORCA engine. Each *worker process* is responsible for an inter-layer partition of the model and can be

placed on a different machine from each other. In particular, each worker manages one or more CPU threads each dedicated for controlling a GPU, the number of which depends on the degree of intra-layer parallelism.

The execution procedure of the ORCA execution engine is as follows. Once the engine is scheduled to run an iteration of the model for a batch of requests, the *engine master* forwards the received information about the scheduled batch to the first *worker process* (Worker1). The information includes tokens for the current iteration and a control message, which is composed of ids of requests within the batch, current token index (for requests in the increment phase), and number of input tokens (for requests in the initiation phase). The *controller* of Worker1 hands over the information received from the engine master to the GPU-controlling threads, where each thread parses the information and issues proper GPU kernels to its associated GPU. For example, the kernel for the Attention operation uses the request id and the current token index to get the GPU memory address of previous keys and values kept by the Attention K/V manager. In the meantime, the controller also forwards the control message to the controller of the next worker (Worker2), without waiting for the completion of the kernels issued on the GPUs of Worker1. Unlike Worker1, the controller of the last worker (Worker2) waits for (i.e., synchronize with) the completion of the issued GPU kernels, in order to fetch the output token for each request and send the tokens back to the engine master.

To keep GPUs busy as much as possible, we design the ORCA engine to minimize synchronization between the CPU and GPUs. We observe that current systems for distributed inference (e.g., FasterTransformer [4] and Megatron-LM [3]) have CPU-GPU synchronization whenever each process receives control messages<sup>5</sup> because they exchange the messages through a GPU-to-GPU communication channel – NCCL [5]. The exchange of these control messages occurs at every iteration, imposing a non-negligible performance overhead. On the other hand, ORCA separates the communication channels for control messages (plus tokens) and tensor data transfer, avoiding the use of NCCL for data used by CPUs. Figure 7 shows that the ORCA engine uses NCCL exclusively for exchanging intermediate tensor data (represented by dashed arrows) as this data is produced and consumed by GPUs. Control messages, which is used by the CPU threads for issuing GPU kernels, sent between the engine master and worker controllers by a separate communication channel that does not involve GPU such as gRPC [2].

## 4.2 Scheduling Algorithm

The ORCA scheduler makes decisions on which requests should be selected and processed at every iteration. The scheduler has high flexibility in selecting a set of requests to com-

<sup>5</sup>This includes various metadata such as batch size, sequence length, and whether a request within the batch has finished processing.

pose a batch, because of the selective batching technique that allows the engine to run any set of requests in the batched manner. Now the main question left is how to select the requests at every iteration.

We design the ORCA scheduler to use a simple algorithm that does not change the processing order of client requests; early-arrived requests are processed earlier. That is, we ensure iteration-level first-come-first-served (FCFS) property. We define the iteration-level FCFS property for workloads with multi-iteration characteristics as follows: for any pair of requests  $(x_i, x_j)$  in the request pool, if  $x_i$  has arrived earlier than  $x_j$ ,  $x_i$  should have run the same or more iterations than  $x_j$ . Note that some late-arrived requests may return earlier to clients if the late request requires a smaller number of iterations to finish.

Still, the scheduler needs to take into account additional factors: diminishing returns to increasing the batch size and GPU memory constraint. Increasing the batch size trades off increased throughput for increased latency, but as the batch size becomes larger, the amount of return (i.e., increase in throughput) diminishes. Therefore, just like other serving systems [7, 16], ORCA also has a notion of a max batch size: the largest possible number of requests within a batch. The ORCA system operator can tune this knob to maximize throughput while satisfying one’s latency budget. We will discuss this in more details with experiment results in Section 6.2.

Another factor is the GPU memory constraint. Optimizing memory usage by reusing buffers for intermediate results across multiple operations is a well-known technique used by various systems [4, 6], and ORCA also adopts this technique. However, unlike the buffers for intermediate results that can be reused immediately, buffers used by the Attention K/V manager for storing the keys and values cannot be reclaimed until the ORCA scheduler notifies that the corresponding request has finished processing. A naïve implementation can make the scheduler fall into a deadlock when the scheduler cannot issue an iteration for any requests in the pool because there is no space left for storing a new Attention key and value for the next token. This requires the ORCA scheduler to be aware of the remaining size of pre-allocated memory regions for the manager.

The ORCA scheduler takes all these factors into account: it selects at most “max batch size” requests based on the arrival time, while reserving enough space for storing keys and values to a request when the request is scheduled for the first time. We describe the scheduling process in Algorithm 1. The algorithm selects a batch of requests from the request pool (line 4) and schedules the batch (line 5). The *Select* function (line 17) selects at most *max\_bs* requests from the pool based on the arrival time of the request (lines 20-22). Algorithm 1 does not depict the procedure of request arrival and return; one may think of it as there exist concurrent threads inserting newly arrived requests into *request\_pool* and removing finished requests from *request\_pool*.

---

**Algorithm 1:** ORCA scheduling algorithm

---

**Params:** *n\_workers*: number of workers, *max\_bs*: max batch size, *n\_slots*: number of K/V slots

```

1 n_scheduled  $\leftarrow 0$ 
2 n_rsrv  $\leftarrow 0$ 
3 while true do
4   batch, n_rsrv  $\leftarrow \text{Select}(\text{request\_pool}, n_{\text{rsrv}})$ 
5   schedule engine to run one iteration of
      the model for the batch
6   foreach req in batch do
7     | req.state  $\leftarrow \text{RUNNING}$ 
8     | n_scheduled  $\leftarrow n_{\text{scheduled}} + 1$ 
9     | if n_scheduled  $= n_{\text{workers}} then
10    |   wait for return of a scheduled batch
11    |   foreach req in the returned batch do
12      |     | req.state  $\leftarrow \text{INCREMENT}$ 
13      |     | if finished(req) then
14        |       | n_rsrv  $\leftarrow n_{\text{rsrv}} - \text{req}.max\_tokens$ 
15        |       | n_scheduled  $\leftarrow n_{\text{scheduled}} - 1$ 
16
17 def Select(pool, n_rsrv):
18   batch  $\leftarrow \{\}$ 
19   pool  $\leftarrow \{\text{req} \in \text{pool} \mid \text{req.state} \neq \text{RUNNING}\}$ 
20   SortByArrivalTime(pool)
21   foreach req in pool do
22     | if batch.size()  $= max\_bs then break
23     | if req.state  $= \text{INITIATION}$  then
24       |   new_n_rsrv  $\leftarrow n_{\text{rsrv}} + \text{req}.max\_tokens$ 
25       |     | if new_n_rsrv  $> n_{\text{slots}} then break
26       |     | n_rsrv  $\leftarrow new\_n_{\text{rsrv}}$ 
27       |     | batch  $\leftarrow batch \cup \{\text{req}\}$ 
28   return batch, n_rsrv$$$ 
```

---

When the scheduler considers a request in the initiation phase, meaning that the request has never been scheduled yet, the scheduler uses the request’s *max\_tokens*<sup>6</sup> attribute to reserve *max\_tokens* slots of GPU memory for storing the keys and values in advance (lines 23-26). The scheduler determines whether the reservation is possible (line 25) based on *n\_rsrv*, the number of currently reserved slots, where a slot is defined by the amount of memory required for storing an Attention key and value for a single token. Here, *n\_slots* is a parameter tuned by the ORCA system operator indicating the size of memory region (in terms of slots) allocated to the Attention K/V manager. Since the number of tokens in a request cannot exceed *max\_tokens*, if the reservation is possible, it is guaranteed that the manager can allocate buffers for the newly generated keys and values until the request finishes.

Unlike the tuning of *max\_bs* that requires quantifying the trade-off between latency and throughput, the ORCA system

---

<sup>6</sup>The *max\_tokens* attribute is a per-request option, meaning the maximum number of tokens that a request can have after processing.

	Time →					
Worker1	A <sub>1</sub> B <sub>1</sub>	C <sub>1</sub> D <sub>1</sub>	E <sub>1</sub> F <sub>1</sub>	A <sub>2</sub> B <sub>2</sub>	C <sub>2</sub> D <sub>2</sub>	E <sub>2</sub> F <sub>2</sub>
Worker2		A <sub>1</sub> B <sub>1</sub>	C <sub>1</sub> D <sub>1</sub>	E <sub>1</sub> F <sub>1</sub>	A <sub>2</sub> B <sub>2</sub>	C <sub>2</sub> D <sub>2</sub>
Worker3			A <sub>1</sub> B <sub>1</sub>	C <sub>1</sub> D <sub>1</sub>	E <sub>1</sub> F <sub>1</sub>	A <sub>2</sub> B <sub>2</sub>

(a) ORCA execution pipeline.

	Time →					
Partition1	A <sub>1</sub>	B <sub>1</sub>	A <sub>2</sub>	B <sub>2</sub>	A <sub>3</sub>	
Partition2		A <sub>1</sub>	B <sub>1</sub>	A <sub>2</sub>	B <sub>2</sub>	
Partition3			A <sub>1</sub>	B <sub>1</sub>	A <sub>2</sub>	B <sub>2</sub>

(b) FasterTransformer execution pipeline.

Figure 8: Comparison of the use of pipeline parallelism in ORCA and FasterTransformer where  $X_i$  is the i-th iteration of request  $X$ .

operator can easily configure  $n\_slots$  without any experiments. Given a model specification (e.g., hidden size, number of layers, etc.) and degrees of intra- and inter-layer parallelism, ORCA’s GPU memory usage mostly depends on  $n\_slots$ . That is, the operator can simply use the largest possible  $n\_slots$  under the memory constraint.

**Pipeline parallelism.** ORCA’s scheduler makes the execution of workers in the engine to be pipelined across multiple batches. The scheduler does not wait for the return of a scheduled batch until  $n\_scheduled$ , the number of currently scheduled batches, reaches  $n\_workers$  (line 9-10 of Algorithm 1). By doing so, the scheduler keeps the number of concurrently running batches in the engine to be  $n\_workers$ , which means that every worker in the engine is processing one of the batches without being idle.

Figure 8a depicts the execution pipeline of 3 ORCA workers, using a max batch size of 2. We assume that the request A arrives before B, which arrives before C, and so on. At first, the scheduler selects requests A and B based on the arrival time and schedules the engine to process a batch of requests A and B (we call this batch AB), where Worker1, Worker2, and Worker3 process the batch in turn. The scheduler waits for the return of the batch AB only after the scheduler injects two more batches: CD and EF. Once the batch AB returns, requests A and B get selected and scheduled once again, because they are the earliest arrived requests among the requests in the pool.

In contrast, the interface between current serving systems and execution engines (e.g., a combination of Triton [7] and FasterTransformer [4]) does not allow injecting another batch before the finish of the current running batch, due to the request-level scheduling. That is, Triton cannot inject the next request C to FasterTransformer until the current

# Params	# Layers	Hidden size	# Inter-partitions	# Intra-partitions
13B	40	5120	1	1
101B	80	10240	1	8
175B	96	12288	2	8
341B	120	15360	4	8

Table 1: Configurations of models used in the experiments.

batch AB finishes. To enable pipelined execution of multiple inter-layer partitions under such constraint, FasterTransformer splits a batch of requests into multiple *microbatches* [28] and pipelines the executions of partitions across the microbatches. In Figure 8b, FasterTransformer splits the batch AB into two microbatches, A and B. Since each partition processes a microbatch (which is smaller than the original batch) in the batched manner, the performance gain from batching can become smaller. Moreover, this method may insert *bubbles* into the pipeline when the microbatch size is too large, making the number of microbatches smaller than the number of partitions. While FasterTransformer needs to trade batching efficiency (larger microbatch size) for pipelining efficiency (fewer pipeline bubbles), ORCA is free of such a tradeoff – thanks to iteration-level scheduling – and can easily pipeline requests without dividing a batch into microbatches.

## 5 Implementation

We have implemented ORCA with 13K lines of C++, based on the CUDA ecosystem. We use gRPC [2] for the communication in the control plane of the ORCA engine, while NCCL [5] is used in the data plane, for both inter-layer and intra-layer communication. Since we design ORCA to focus on Transformer-based generative models, ORCA provides popular Transformer layers as a building block of models including the original encoder-decoder Transformer [60], GPT [47], and other variants discussed in Raffel et al. [49].

We have also implemented fused kernels for LayerNorm, Attention, and GeLU operators, just like other systems for training or inference of Transformer models [1, 4, 58]. For example, the procedure of computing dot products between Attention query and keys, Softmax on the dot products, and weighted average of Attention values are fused into a single CUDA kernel for the Attention operator. In addition, we go one step further and fuse the kernels of the split Attention operators by simply concatenating all thread blocks of the kernels for different requests. Although this fusion makes the thread blocks within a kernel have different characteristics and lifetimes (which is often discouraged by CUDA programming practice) because they process tensors of different shapes, we find this fusion to be beneficial by improving GPU utilization and reducing the kernel launch overhead [34, 39].

## 6 Evaluation

In this section, we present evaluation results to show the efficiency of ORCA.

**Environment.** We run our evaluation on Azure ND96asr A100 v4 VMs, each equipped with 8 NVIDIA 40-GB A100 GPUs connected over NVLink. We use at most four VMs depending on the size of the model being tested. Each VM has 8 Mellanox 200Gbps HDR Infiniband adapters, providing an 1.6Tb/s of interconnect bandwidth between VMs.

**Models.** Throughout the experiments, we use GPT [12] as a representative example of Transformer-based generative models. We use GPT models with various configurations, which is listed in Table 1. The configurations for 13B and 175B models come from the GPT-3 paper [12]. Based on these two models, we change the number of layers and hidden size to make configurations for 101B and 341B models. All models have a maximum sequence length of 2048, following the setting of the original literature [12]. We use fp16-formatted model parameters and intermediate activations for the experiments. We also apply inter- and intra-layer parallelism strategies described in Section 4.1, except for the 13B model that can fit in a GPU. For example, the 175B model is partitioned over a total of 16 GPUs by using 2 inter-layer partitions subdivided into 8 intra-layer partitions, where the 8 GPUs in the same VM belongs to the same inter-layer partition.

**Baseline system.** We compare with FasterTransformer [4], an inference engine that supports large scale Transformer models via distributed execution. While there exist other systems with the support for distributed execution such as Megatron-LM [3] and DeepSpeed [1], these systems are primarily designed and optimized for training workloads, which makes them show relatively lower performance compared to the inference-optimized systems.

**Scenarios.** We use two different scenarios to drive our evaluation. First, we design a microbenchmark to solely assess the performance of the ORCA engine without being affected by the iteration-level scheduling. In particular, we do not run the ORCA scheduler in this scenario. Instead, given a batch of requests, the testing script repeats injecting the same batch into the ORCA engine until all requests in the batch finishes, mimicking the behavior of the canonical request-level scheduling. We also assume that all requests in the batch have the same number of input tokens and generate the same number of output tokens. We report the time taken for processing the batch (not individual requests) and compare the result with FasterTransformer [4].

The second scenario tests the end-to-end performance of ORCA by emulating a workload. We synthesize a trace of

client requests because there is no publicly-available request trace for generative language models. Each request in the synthesized trace is randomly generated by sampling the number of input tokens and a *max\_gen\_tokens* attribute, where the number of input tokens plus *max\_gen\_tokens* equals to the *max\_tokens* attribute described in Section 4.2. We assume that all requests continue generation until the number of generated tokens reaches *max\_gen\_tokens*. In other words, we make the model never emit the “<EOS>” token. This is because we have neither the actual model checkpoint nor the actual input text so we do not have any information to guess the right timing of the “<EOS>” token generation. Once the requests are generated, we synthesize the trace by setting the request arrival time based on the Poisson process. To assess ORCA’s behavior under varying load, we change the Poisson parameter (i.e., arrival rate) and adjust the request arrival time accordingly. We report latency and throughput using multiple traces generated from different distributions for better comparison and understanding of the behavior of ORCA and FasterTransformer.

### 6.1 Engine Microbenchmark

We first compare the performance of FasterTransformer and the ORCA engine using the first scenario. We set all requests in the batch to have the same number of input tokens (32 or 128) and generate 32 tokens. That is, in this set of experiments, all requests within the batch start and finish processing at the same time. We conduct experiments using three different models: 13B, 101B, and 175B. For each model, we use the corresponding parallelization strategy shown in Table 1.

Figure 9 shows the performance of FasterTransformer and the ORCA engine for processing a batch composed of the same requests. In Figure 9a, the ORCA engine shows a similar (or slightly worse) performance compared to FasterTransformer across all configurations. This is because ORCA does not apply batching to the Attention operations, while FasterTransformer apply batching to all operations. Still, the performance difference is relatively small. Despite not batching the Attention operation, the absence of model parameters in Attention makes this decision has little impact on efficiency as there is no benefit of reusing model parameters across multiple requests.

Figure 9b presents similar results for the 101B model that uses all of the 8 GPUs in a single VM. From these results, we can say that the ORCA engine and FasterTransformer have comparable efficiencies in the implementations of CUDA kernels and the communication between intra-layer partitions. Note that FasterTransformer cannot use a batch size of 8 or larger with the 13B model (16 or larger with the 101B model) because of the fixed amount of memory pre-allocation for each request’s Attention keys and values, which grows in proportion to the max sequence length of the model (2048 for this case). In contrast, ORCA avoids redundant memory

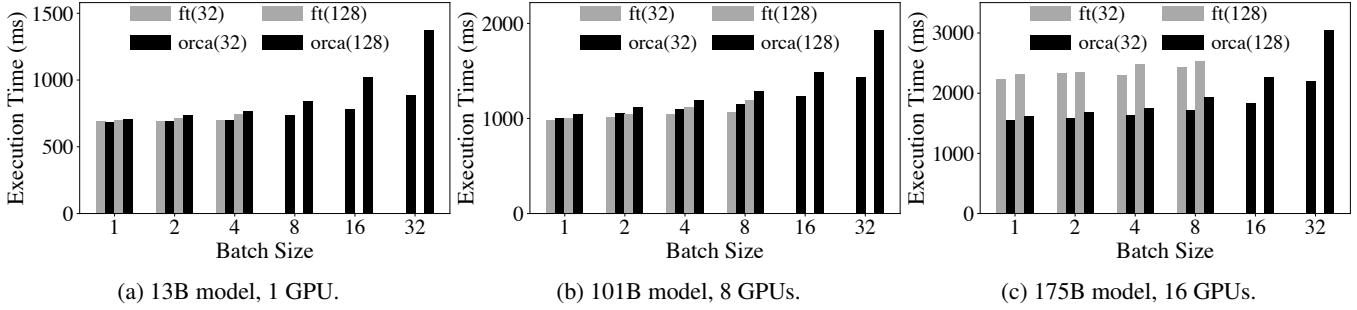


Figure 9: Execution time of a batch of requests using FasterTransformer and the ORCA engine without the scheduling component. Label “ft( $n$ )” represents results from FasterTransformer processing requests with  $n$  input tokens. Configurations that incur out of memory error are represented as missing entries (e.g., ft(32) for the 101B model with a batch size of 16).

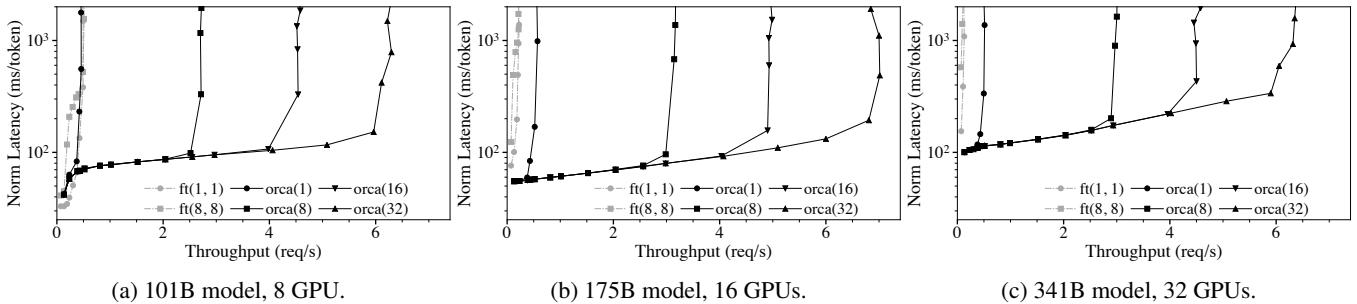


Figure 10: Median end-to-end latency normalized by the number of generated tokens and throughput. Label “orca( $max\_bs$ )” represents results from ORCA with a max batch size of  $max\_bs$ . Label “ft( $max\_bs, mbs$ )” represents results from FasterTransformer with a max batch size of  $max\_bs$  and a microbatch size of  $mbs$ .

allocation by setting the size of buffers for the keys and values separately for each request based on the *max\_tokens* attribute.

Next, we go one step further and experiment with the 175B model, which splits the layers into two inter-layer partitions. In this case, for better comparison, we disable pipelined execution of the inter-layer partitions for both systems. For FasterTransformer, we set the size of a microbatch to be equal to the batch size to disable pipelining. As shown in Figure 9c, the ORCA engine outperforms FasterTransformer by up to 47%. We attribute this performance improvement to the control-data plane separation described in Section 4.1. We omit the 341B model as it has similar results compared to the 175B model.

## 6.2 End-to-end Performance

Now we assess the end-to-end performance of ORCA by measuring the latency and throughput with the synthesized request trace under varying load. When synthesizing the trace, we sample each request’s number of input tokens from  $U(32, 512)$ , a uniform distribution ranging from 32 to 512 (inclusive). The *max\_gen\_tokens* attributed is sampled from  $U(1, 128)$ , which means that the least and the most time-consuming requests require 1 and 128 iterations of the model for processing, respectively.

Unlike the microbenchmark shown in Section 6.1, to measure the end-to-end performance, we test the entire ORCA software stack including the ORCA scheduler. Client requests arrive to the ORCA scheduler following the synthesized trace described above. We report results from various max batch size configurations. For FasterTransformer that does not have its own scheduler, we implement a custom scheduler that receives client requests, creates batches, and injects the batches to an instance of FasterTransformer. We make the custom scheduler create batches dynamically by taking at most max batch size requests from the request queue, which is the most common scheduling algorithm used by existing serving systems like Triton [7] and TensorFlow Serving [42]. Again, we report results from various max batch size configurations, along with varying microbatch sizes, an additional knob in FasterTransformer that governs the pipelining behavior (see Section 4.2).

Figure 10 shows median end-to-end latency and throughput. Since each request in the trace requires different processing time, which is (roughly) in proportion to the number of generated tokens, we report median latency normalized by the number of generated tokens of each request. From the figure, we can see that ORCA provides significantly higher throughput and lower latency than FasterTransformer. The only exception is the 101B model under low load (Figure 10a). In this

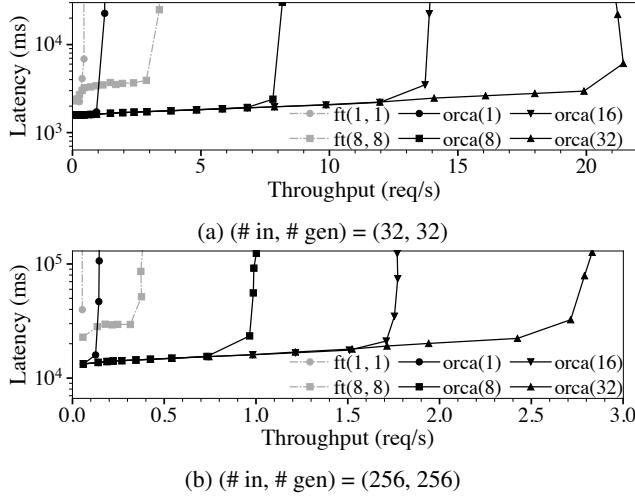


Figure 11: Median end-to-end latency and throughput, using the 175B model with traces composed of homogeneous requests. We do not normalize the latency since all requests have the same characteristic.

case, both ORCA and FasterTransformer do not have enough number of requests to process in a batch. That is, the latency will mostly depend on the engine’s performance, which is shown in Figure 9b. As the load becomes heavier, ORCA provides higher throughput with a relatively small increase in latency, because the ORCA scheduler makes late-arrived requests hitch a ride with the current ongoing batch. In contrast, FasterTransformer fails to efficiently handle multiple requests that (1) arrive at different times; (2) require different number of iterations to finish; or (3) start with different number of input tokens, resulting in a peak throughput of 0.49 req/s and much higher latency. If we use the 175B or 341B model (Figures 10b and 10c) that employs more than one inter-layer partitions, ORCA outperforms FasterTransformer under every level of load in terms of both latency and throughput, resulting in an order of magnitude higher throughput when we compare results at a similar level of latency. For example, to match a median normalized latency of 190ms for the 175B model, which is a double of the normalized execution time (by the number of generated tokens) of “orca(128)” shown in Figure 9c, FasterTransformer provides a throughput of 0.185 req/s whereas ORCA provides a throughput of 6.81 req/s, which is a  $36.9 \times$  speedup.

**Varying batch size configurations.** Figure 10 shows that the increase of the max batch size of ORCA results in a higher throughput without affecting the latency. This is because the iteration-level scheduling of ORCA resolves the problem of early-finished and late-joining requests. Nevertheless, there is no guarantee that increasing the batch size will not negatively affect the latency, for arbitrary hardware settings, models, and workloads. As mentioned in Section 4.2, the max batch size

must be set carefully by considering both the required latency and throughput requirements.

Interestingly, larger max batch size in FasterTransformer does not necessarily help improving throughput. By testing all possible combinations of max batch size ( $max\_bs$ ) and microbatch size ( $mbs$ ) on all models under varying load, we find that  $(max\_bs, mbs) = (1, 1)$  or  $(8, 8)$  are the best options. Per our discussion in Section 4.1, FasterTransformer’s microbatch-based pipelining can be less efficient because the engine is going to process at most  $mbs$  number of requests in the batched manner, which explains why the configurations with the maximum possible  $mbs$  (which is the same as  $max\_bs$ ) have better performance than others. In addition, while increasing  $max\_bs$  can improve performance due to the increased batch size, at the same time, this also increases the likelihood of batching requests with large difference in the number of input tokens or the number of generated tokens. In such cases, FasterTransformer cannot efficiently handle the batch because (1) for the first iteration of the batch, FasterTransformer processes requests as if they all had the same input length as the shortest one; and (2) early-finished requests cannot immediately return to the clients.

**Trace of homogeneous requests.** We test the behavior of ORCA and FasterTransformer when using a trace of homogeneous requests, i.e., all requests in a trace have the same number of input tokens and the same  $max\_gen\_tokens$  attribute. Since all requests require the same number of iterations to finish processing, the problem of early-leaving requests does not occur for this trace. As a result, now the increase of the  $max\_bs$  has a noticeable positive impact on the performance of FasterTransformer, as shown in Figure 11. Still, ORCA outperforms FasterTransformer ( $max\_bs=8$ ) except for the case using a max batch size of 1, where ORCA degenerates into a simple pipeline of the ORCA workers that does not perform batching.

## 7 Related Work and Discussion

**Fine-grained batching for recurrent models.** We would like to highlight BatchMaker [23] as one of the most relevant previous works. BatchMaker is a serving system for RNNs that performs scheduling and batching at the granularity of RNN cells, motivated by the unique RNN characteristic of repeating the same computation. Once a request arrives, BatchMaker breaks the dataflow graph for processing the request into RNN cells, schedules execution at the granularity of cells (instead of the entire graph), and batches the execution of identical cells (if any). Since each RNN cell always performs the exact same computation, BatchMaker can execute multiple RNN cells in a batched manner regardless of the position (i.e., token index) of the cell. By doing so, BatchMaker allows a newly arrived request for RNN to join (or a finished request

to leave) the current executing batch without waiting for the batch to completely finish.

However, BatchMaker cannot make batches of cells for Transformer models because there are too many distinct cells (a subgraph that encapsulates the computation for processing a token; Figure 1c) in the graph. Each cell at a different token index  $t$  must use a different set of Attention Keys/Values. As the cell for each  $t$  is different, the graph comprises  $L$  different cells ( $L$  denotes the number of input and generated tokens), significantly lowering the likelihood of cells of the same computation being present at a given moment (e.g., in Figure 10,  $L$  ranges from  $33 = 32 + 1$  to  $640 = 512 + 128$ ). Thus execution of the cells will be mostly serialized, making BatchMaker fall back to non-batched execution. BatchMaker also lacks support for large models that require model and pipeline parallelism.

While BatchMaker is geared towards detecting and aligning batch-able RNN cells, our key principle in designing ORCA is to perform as much computation as possible per each round of model parameter read. This is based on the insight that reading parameters from GPU global memory is a major bottleneck in terms of end-to-end execution time, for large-scale models. Adhering to this principle, we apply iteration-level scheduling and selective batching to process all “ready” tokens in a single round of parameter read, regardless of whether the processing of tokens can be batched (non-Attention ops) or not (Attention ops).

**Specialized execution engines for Transformer models.** The outstanding performance of Transformer-based models encourages the development of inference systems specialized for them. FasterTransformer [4], LightSeq [61], TurboTransformers [22] and EET [36] are such examples. Each of these systems behave as an backend execution engine of existing serving systems like Triton Inference Server [7] and TensorFlow Serving [42]. That is, these systems delegate the role of scheduling to the serving system layer, adhering to the canonical request-level scheduling. Instead, ORCA suggests to schedule executions at a finer granularity, which is not possible in current systems without changing the mechanism for coordination between the scheduler and the execution engine. Note that among these systems, FasterTransformer is the only one with the support for distributed execution. While systems like Megatron-LM [3] and DeepSpeed [1] can also be used for distributed execution, these systems are primarily optimized for large-scale training rather than inference serving.

**Interface between serving systems and execution engines.** Current general-purpose serving systems such as Triton Inference Server [7] and Clipper [16] serve as an abstraction for handling client requests and scheduling executions of the underlying execution engines. This approach is found to be beneficial by separating the design and implementation of the serving layer and the execution layer. However, we find

that the prevalent interface between the two layers is too restricted for handling models like GPT [12], which has the multi-iteration characteristic. Instead, we design ORCA to tightly integrate the scheduler and the engine, simplifying the application of the two proposed techniques: iteration-level scheduling and selective batching. While in this paper we do not study a general interface design that supports the two techniques without losing the separation of abstractions, it can be an interesting topic to explore such possibility; we leave this issue to future work.

## 8 Conclusion

We present iteration-level scheduling with selective batching, a novel approach that achieves low latency and high throughput for serving Transformer-based generative models. Iteration-level scheduling makes the scheduler interact with the execution engine at the granularity of iteration instead of request, while selective batching enables batching arbitrary requests processing tokens at different positions, which is crucial for applying batching with iteration-level scheduling. Based on these techniques, we have designed and implemented a distributed serving system named ORCA. Experiments show the effectiveness of our approach: ORCA provides an order of magnitude higher throughput than current state-of-the-art systems at the same level of latency.

## Acknowledgments

We thank our shepherd Amar Phanishayee and the anonymous reviewers for their insightful comments. This work was supported by FriendliAI Inc.

## References

- [1] DeepSpeed. Retrieved Dec 13, 2021 from <https://github.com/microsoft/DeepSpeed>.
- [2] gRPC. Retrieved Dec 13, 2021 from <https://grpc.io>.
- [3] Megatron-LM. Retrieved Dec 13, 2021 from <https://github.com/NVIDIA/Megatron-LM>.
- [4] NVIDIA FasterTransformer. Retrieved Dec 13, 2021 from <https://github.com/NVIDIA/FasterTransformer>.
- [5] NVIDIA NCCL. Retrieved Dec 13, 2021 from <https://github.com/NVIDIA/nccl>.
- [6] NVIDIA TensorRT. Retrieved Dec 13, 2021 from <https://developer.nvidia.com/tensorrt>.

- [7] NVIDIA Triton Inference Server. Retrieved Dec 13, 2021 from <https://developer.nvidia.com/nvidia-triton-inference-server>.
- [8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pages 265–283, 2016.
- [9] Daniel Adiwardana, Minh-Thang Luong, David R So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, et al. Towards a Human-like Open-Domain Chatbot. *arXiv preprint arXiv:2001.09977*, 2020.
- [10] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mi-haylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, Giri Anantharaman, Xian Li, Shuhui Chen, Halil Akin, Mandeep Baines, Louis Martin, Xing Zhou, Punit Singh Koura, Brian O’Horo, Jeff Wang, Luke Zettlemoyer, Mona Diab, Zornitsa Kozareva, and Ves Stoyanov. Efficient Large Scale Language Modeling with Mixtures of Experts. *arXiv preprint arXiv:2112.10684*, 2021.
- [11] Peter F. Brown, John Cocke, Stephen A. Della Pietra, Vincent J. Della Pietra, Fredrick Jelinek, John D. Lafferty, Robert L. Mercer, and Paul S. Roossin. A Statistical Approach to Machine Translation. *Computational Linguistics*, 16(2):79–85, 1990.
- [12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 2020.
- [13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebbgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, 2021.
- [14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, pages 579–594, 2018.
- [15] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeon-taeck Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zong-wei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling Language Modeling with Pathways. *arXiv preprint arXiv:2204.02311*, 2022.
- [16] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, pages 613–627, 2017.

- [17] Raj Dabre, Chenhui Chu, and Anoop Kunchukuttan. A Survey of Multilingual Neural Machine Translation. *ACM Computing Surveys*, 53(5), 2020.
- [18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.
- [19] Ming Ding, Zhuoyi Yang, Wenyi Hong, Wendi Zheng, Chang Zhou, Da Yin, Junyang Lin, Xu Zou, Zhou Shao, Hongxia Yang, and Jie Tang. CogView: Mastering Text-to-Image Generation via Transformers. *Advances in Neural Information Processing Systems*, 2021.
- [20] Lucas Dixon, John Li, Jeffrey Sorensen, Nithum Thain, and Lucy Vasserman. Measuring and Mitigating Unintended Bias in Text Classification. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, pages 67–73, 2018.
- [21] Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellar, Kevin Robinson, Kathy Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. GLaM: Efficient Scaling of Language Models with Mixture-of-Experts. *arXiv preprint arXiv:2112.06905*, 2021.
- [22] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. TurboTransformers: An Efficient GPU Serving System for Transformer Models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.
- [23] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low Latency RNN Inference with Cellular Batching. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.
- [24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.
- [25] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.
- [26] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Milligan, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training Compute-Optimal Large Language Models. *arXiv preprint arXiv:2203.15556*, 2022.
- [27] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, pages 269–286, 2018.
- [28] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. *Advances in Neural Information Processing Systems*, 2019.
- [29] Angela H. Jiang, Daniel L.-K. Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A. Kozuch, Padmanabhan Pillai, David G. Andersen, and Gregory R. Ganger. Mainstream: Dynamic Stem-Sharing for Multi-Tenant Video Processing. In *Proceedings of the 2018 USENIX Annual Technical Conference*, pages 29–42, 2018.
- [30] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. NoScope: Optimizing Neural Network Queries over Video at Scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.
- [31] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*, 2020.
- [32] Daniel Khashabi, Sewon Min, Tushar Khot, Ashish Sabharwal, Oyvind Tafjord, Peter Clark, and Hannaneh Hajishirzi. UNIFIEDQA: Crossing Format Boundaries with a Single QA System. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1896–1907, 2020.
- [33] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural Questions: a Benchmark for Question Answering Research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019.
- [34] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and Parallel GPU Task

- Scheduling for Deep Learning. *Advances in Neural Information Processing Systems*, 2020.
- [35] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, pages 611–626, 2018.
- [36] Gongzheng Li, Yadong Xi, Jingzhen Ding, Duan Wang, Bai Liu, Changjie Fan, Xiaoxi Mao, and Zeng Zhao. Easy and Efficient Transformer: Scalable Inference Solution For large NLP model. *arXiv preprint arXiv:2104.12470*, 2021.
- [37] Opher Lieber, Or Sharir, Barak Lenz, and Yoav Shoham. Jurassic-1: Technical details and evaluation. 2021.
- [38] Xudong Lin, Gedas Bertasius, Jue Wang, Shih-Fu Chang, Devi Parikh, and Lorenzo Torresani. Vx2text: End-to-end learning of video-based text generation from multimodal inputs. In *Proceedings of the 2021 IEEE Conference on Computer Vision and Pattern Recognition*, pages 7005–7015, 2021.
- [39] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lin-tao Zhang, and Lidong Zhou. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks, pages 881–897. 2020.
- [40] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a Suit of Armor Conduct Electricity? A New Dataset for Open Book Question Answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2381–2391, 2018.
- [41] Ramesh Nallapati, Bowen Zhou, Cícero Nogueira dos Santos, Çağlar Gülcöhre, and Bing Xiang. Abstractive Text Summarization using Sequence-to-sequence RNNs and Beyond. In *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning*, pages 280–290, 2016.
- [42] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. TensorFlow-Serving: Flexible, High-Performance ML Serving. *Workshop on Machine Learning Systems at NIPS 2017*, 2017.
- [43] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, 2019.
- [44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, 2019.
- [45] Romain Paulus, Caiming Xiong, and Richard Socher. A Deep Reinforced Model for Abstractive Summarization. In *Proceedings of the 6th International Conference on Learning Representations*, 2018.
- [46] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.
- [47] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019.
- [48] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saftron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d’Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Ed Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem

- Ayoub, Jeff Stanway, Lorrain Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling Language Models: Methods, Analysis & Insights from Training Gopher. *arXiv preprint arXiv:2112.11446*, 2021.
- [49] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.
- [50] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale. *arXiv preprint arXiv:2201.05596*, 2022.
- [51] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-Shot Text-to-Image Generation. In *Proceedings of the 38th International Conference on Machine Learning*, pages 8821–8831, 2021.
- [52] Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Eric Michael Smith, Y-Lan Boureau, and Jason Weston. Recipes for Building an Open-Domain Chatbot. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 300–325, 2021.
- [53] Timo Schick and Hinrich Schütze. Exploiting Cloze-Questions for Few-Shot Text Classification and Natural Language Inference. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 255–269, 2021.
- [54] Abigail See, Peter J. Liu, and Christopher D. Manning. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1073–1083, 2017.
- [55] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. MeshTensorFlow: Deep Learning for Supercomputers. *Advances in Neural Information Processing Systems*, 2018.
- [56] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.
- [57] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast Video Classification via Adaptive Cascading of Deep Models. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3646–3654, 2017.
- [58] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [59] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *arXiv preprint arXiv:2201.11990*, 2022.
- [60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. *Advances in Neural Information Processing Systems*, 2017.
- [61] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. LightSeq: A High Performance Inference Library for Transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*, pages 113–120, 2021.
- [62] Zihao Wang, Wei Liu, Qian He, Xinglong Wu, and Zili Yi. Clip-gen: Language-free training of a text-to-image generator with clip. *arXiv preprint arXiv:2203.00386*, 2022.
- [63] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. Finetuned Language Models are Zero-Shot Learners. In *Proceedings of the 10th International Conference on Learning Representations*, 2022.
- [64] Ronald J. Williams and David Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, 1(2):270–280, 1989.
- [65] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, Attend and Tell: Neural Image

- Caption Generation with Visual Attention. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 2048–2057, 2015.
- [66] Zhilin Yang, Ye Yuan, Yuexin Wu, William W. Cohen, and Ruslan R. Salakhutdinov. Review Networks for Caption Generation. *Advances in Neural Information Processing Systems*, 2016.

- [67] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuhui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models. *arXiv preprint arXiv:2205.01068*, 2022.

---

# Efficient LLM Inference on CPUs

---

**Haihao Shen Hanwen Chang Bo Dong Yu Luo Hengyu Meng**  
 {haihao.shen, hanwen.chang, bo1.dong, yu.luo, hengyu.meng}@intel.com

## Abstract

Large language models (LLMs) have demonstrated remarkable performance and tremendous potential across a wide range of tasks. However, deploying these models has been challenging due to the astronomical amount of model parameters, which requires a demand for large memory capacity and high memory bandwidth. In this paper, we propose an effective approach that can make the deployment of LLMs more efficiently. We support an automatic INT4 weight-only quantization flow and design a special LLM runtime with highly-optimized kernels to accelerate the LLM inference on CPUs. We demonstrate the general applicability of our approach on popular LLMs including Llama2, Llama, GPT-NeoX, and showcase the extreme inference efficiency on CPUs. The code is publicly available at: <https://github.com/intel/intel-extension-for-transformers>.

## 1 Introduction

Large language models (LLMs) have shown remarkable performance and tremendous potential across a wide range of tasks Rozière et al. [2023], Touvron et al. [2023b,a], Zhang et al. [2022], Brown et al. [2020], Li et al. [2023]. However, deploying these models has been challenging due to the astronomical amount of model parameters, which necessitates significant memory capacity and high memory bandwidth.

Quantization is a technique to reduce the numeric precision of weights and activations of a neural network to lower the computation costs of inference. INT8 quantization Vanhoucke et al. [2011], Han et al. [2015], Jacob et al. [2018] is the most widely-used approach today given the trade-off between high inference performance and reasonable model accuracy. However, outliers in activations have been observed and those outlier values are limiting the general adoption of INT8 quantization, though there are some related work that has been proposed to address the issues Xiao et al. [2023], Wei et al. [2023], Dettmers et al. [2022]. FP8 is a newly introduced data type that has attracted lots of attentions Micikevicius et al. [2022], Kuzmin et al. [2022], Sun et al. [2019], Shen et al. [2023] while it has little adoptions due to the hardware unavailability. On the other hand, weight-only quantization becomes popular as it applies the low precision (e.g., 4-bit) to weights only, while keeping higher precision (e.g., 16-bit floating point) for activations, therefore maintaining the model accuracy. There are many excellent work on 4-bit weight-only quantization Dettmers et al. [2023], Frantar et al. [2022], Cheng et al. [2023b], Lin et al. [2023], Kim et al. [2023], Wu et al. [2023], Cheng et al. [2023a] that have demonstrated the effectiveness in LLM inference. Meanwhile, the open-source community is embracing such low-bit weight-only quantization and offers the CPP-based implementations such as llama.cpp and starcoder.cpp based on ggml library. These implementations are typically optimized for CUDA and may not work on CPUs. Therefore, it is important to address the challenge of making LLM inference efficient on CPU.

In this paper, we propose an effective approach for LLM inference on CPUs including an automatic INT4 quantization flow and an efficient LLM runtime. We leverage Intel Neural Compressor that provides the support of INT4 quantization such as GPTQ Frantar et al. [2022], AWQ Lin et al. [2023], TEQ Cheng et al. [2023a], SignRound Cheng et al. [2023b] and generate the INT4 model automatically. Inspired from the ggml library, we develop a tensor library for CPU, supporting all

the mainstream instruction sets such as AVX2, AVX512, AVX512\_VNNI Rodriguez et al. [2018], and AMX (Advanced Matrix Extensions). Our results show the average latency of generation tokens from 20ms to 80ms on LLMs with 6B to 20B parameters using just a single socket of 4th Generation Intel® Xeon® Scalable Processors, while preserving the high accuracy within only 1% loss from FP32 baseline. Our main contributions are as follows:

- We propose an automatic INT4 quantization flow and generate the high-quality INT4 models with negligible accuracy loss within <1% from FP32 baseline.
- We design a tensor library that supports general CPU instruction sets and latest instruction sets for deep learning acceleration. With CPU tensor library, we develop an efficient LLM runtime to accelerate the inference.
- We apply our inference solution to popular LLM models covering 3B to 20B and demonstrate the promising per-token generation latency from 20ms to 80ms, much faster than the average human reading speed about 200ms per token.

The rest of this paper is organized as follows. Section 2 introduces the approach including INT4 quantization and inference. Section 3 outlines the experimental setup, presents accuracy & performance results, and offers discussion on performance tuning. Section 4 presents the conclusions and future work.

## 2 Approach

In this section, we introduce the approach which consists of two major components: an automatic INT4 quantization flow and an efficient LLM runtime, as shown in Figure 1. More details are described in the following sections.

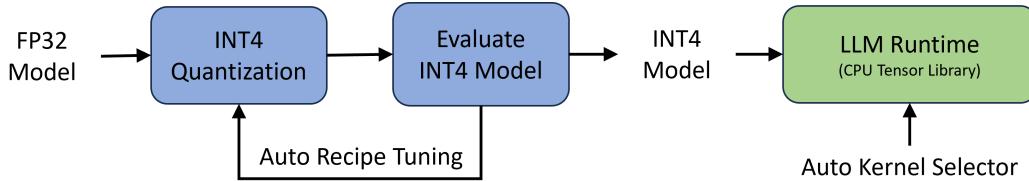


Figure 1: The left part is the automatic INT4 quantization flow: given a FP32 model, the flow takes the default INT4 quantization recipes and evaluates the accuracy of INT4 model; the recipe tuning loop is optional, if INT4 model can meet the accuracy target. The right part is a simplified runtime for efficient LLM inference built on top of a CPU tensor library with automatic kernel selector.

### 2.1 Automatic INT4 Quantization Flow

INT4 quantization flow is developed based on Intel Neural Compressor, a popular quantization tool for deep learning frameworks. Since the tool has already supported the mainstream INT4 quantization recipes such as GPTQ, SignRound, AWQ, TEQ, and RTN (round-to-nearest), our automatic quantization flow allows the recipe tuning on different quantization recipes, different granularities (channel-wise or group-wise), different group size (32, 64, 128 ... 1024). Each recipe generates an INT4 model that is evaluated in the flow. Once the INT4 model meets the accuracy target, the model will be passed to LLM Runtime for performance evaluation.

### 2.2 Efficient LLM Runtime

LLM runtime is designed to provide the efficient inference of LLMs on CPUs. Figure 2 describes the key components in LLM runtime, where the components (CPU tensor library and LLM optimizations) in green are specialized for LLM inference, while the other components (memory management, thread scheduler, operator optimization and fusion) in blue are required for a general runtime. More details about CPU tensor library and LLM optimizations are described in the following paragraphs, while the general components are omitted due to the space limitations. Note that the design is flexibly extensible with hardware abstraction layer (CPU only for now), while how to support other hardware is out of scope in this paper.

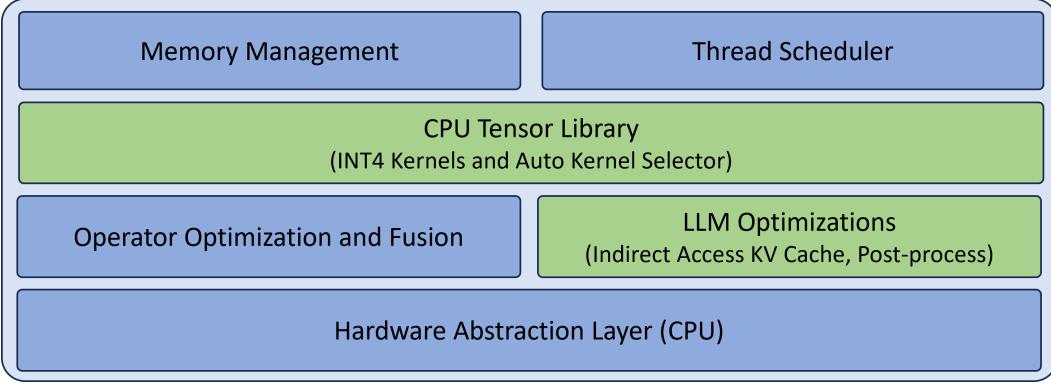


Figure 2: Key components in LLM runtime: general and LLM specialized.

**CPU Tensor Library.** We develop CPU tensor library for linear algebra subroutines, inspired from the template design of cutlass. The tensor library offers a comprehensive support of INT4 kernels for x86 CPUs as shown in Table 1, where AMX is available in the latest Intel Xeon Scalable Processors and VNNI is available in both Intel and AMD CPUs.

Table 1: Support matrix by CPU tensor library: input/output data type, compute data type, and ISA (instruction set architecture). The library supports dynamic quantization for input along with batch or input channel per group, and weight quantization in both symmetric and asymmetric scheme.

Input Data Type	Output Data Type	Compute Data Type	Compute ISA
FP32	FP32	FP32	AVX2
FP32	FP32	FP32	AVX512F
FP32	FP32	INT8	AVX_VNNI
FP32	FP32	INT8	AVX512_VNNI
FP32	FP32	INT8	AMX_INT8
FP32/FP16	FP32/FP16	FP16	AVX512_FP16
FP32/BF16	FP32/BF16	BF16	AMX_BF16

**LLM Optimizations.** Most recent LLMs are typically decoder-only Transformer-based models Vaswani et al. [2017]. Given the unique characteristics of next token generation, KV cache becomes performance critical for LLM inference. We describe the optimizations in Figure 3.

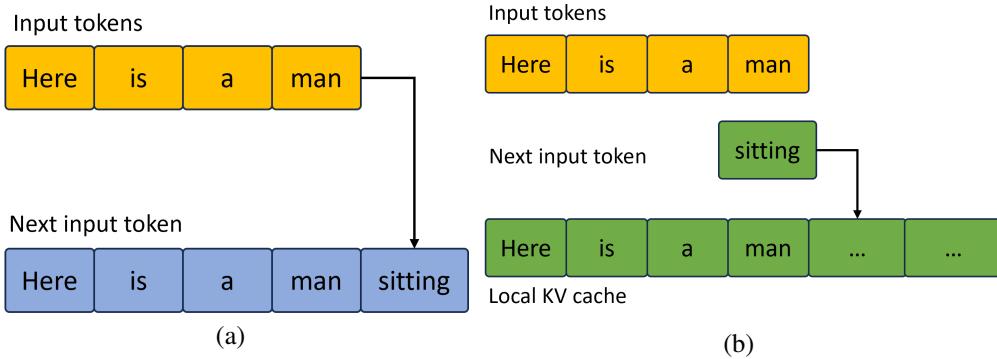


Figure 3: KV cache optimization. Left (a) shows the default KV cache, where new token generation requires memory reallocation for all the tokens (5 in this example); right (b) shows the optimized KV cache with pre-allocated KV memory and only new token updated each time.

### 3 Results

#### 3.1 Experimental Setup

To demonstrate the generality, we select the popular LLMs across a wide range of architectures with the model parameter size from 7B to 20B. We evaluate the accuracy of both FP32 and INT4 models using open-source datasets from lm-evaluation-harness including lambada Paperno et al. [2016] openai, hellaswag Zellers et al. [2019], winogrande Sakaguchi et al. [2021], piqa Bisk et al. [2020], and wikitext. To demonstrate the performance, we measure the latency of next token generation on the 4th Generation Intel® Xeon® Scalable Processors, available on the public clouds such as AWS.

#### 3.2 Accuracy

We evaluate the accuracy on the aforementioned datasets and show the average accuracy in Table 2. We can see from the table that the accuracy of INT4 model is nearly on par with that of FP32 model within 1% relative loss from FP32 baseline.

Table 2: INT4 and FP32 model accuracy. INT4 model has two configurations: group size=4 and 128.

LLM	FP32	INT4 (Group size=32)	INT4 (Group size=128)
EleutherAI/gpt-j-6B	0.643	0.644	0.64
meta-llama/Llama-2-7b-hf	0.69	0.69	0.685
decapoda-research/llama-7b-hf	0.689	0.682	0.68
EleutherAI/gpt-neox-20b	0.674	0.672	0.669
tiuae/falcon-7b	0.698	0.694	0.693

#### 3.3 Performance

We measure the latency of next token generation using LLM runtime and the popular open-source ggml-based implementation. Table 3 presents the latency under a proxy configuration with 32 as both input and output tokens. Note that ggml-based solution only supports group size 32 when testing.

Table 3: INT4 performance using LLM runtime and ggml-based solution. LLM runtime outperforms ggml-based solution by up to 1.6x under group-size=128 and 1.3x under group size=32.

model	LLM Runtime (Group size=32)	LLM Runtime (Group size=128)	ggml-based (Group size=32)
EleutherAI/gpt-j-6B	22.99ms	19.98ms	31.62ms
meta-llama/Llama-2-7b-hf	23.4ms	21.96ms	27.71ms
decapoda-research/llama-7b-hf	23.88ms	22.04ms	27.2ms
EleutherAI/gpt-neox-20b	80.16ms	61.21ms	92.36ms
tiuae/falcon-7b	31.23ms	22.26ms	36.22ms

#### 3.4 Discussion

Though we demonstrate the performance advantage over ggml-based solution, there are still opportunities for LLM runtime to further improve the performance through additional performance tuning such as thread scheduler in LLM runtime, blocking strategy in CPU tensor library.

## 4 Summary and Future Work

We presented an end-to-end INT4 LLM inference including an automatic INT4 model quantization and efficient LLM runtime. We demonstrated the generality on a set of popular LLMs and the performance advantage over the open-source solution on CPUs. As our future works, we plan to further improve the CPU tensor library and extend Hugging Face transformer APIs to support INT4 LLM inference as part of the contributions to the open-source community. Moreover, we plan to exercise our approach on personal computers (PCs) given the broad accessibility of CPUs, to meet the growing demands of AI generated content and empower generative AI on PCs.

## References

- Y. Bisk, R. Zellers, J. Gao, Y. Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439, 2020.
- T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- W. Cheng, Y. Cai, K. Lv, and H. Shen. Teq: Trainable equivalent transformation for quantization of llms. *arXiv preprint arXiv:2310.10944*, 2023a.
- W. Cheng, W. Zhang, H. Shen, Y. Cai, X. He, and K. Lv. Optimize weight rounding via signed gradient descent for the quantization of llms. *arXiv preprint arXiv:2309.05516*, 2023b.
- T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.
- T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.
- E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.
- S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2015. URL <https://arxiv.org/abs/1510.00149>.
- B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.
- S. Kim, C. Hooper, A. Gholami, Z. Dong, X. Li, S. Shen, M. W. Mahoney, and K. Keutzer. Squeezellm: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*, 2023.
- A. Kuzmin, M. Van Baalen, Y. Ren, M. Nagel, J. Peters, and T. Blankevoort. Fp8 quantization: The power of the exponent. *Advances in Neural Information Processing Systems*, 35:14651–14662, 2022.
- R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.
- J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.
- P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu, et al. Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.
- D. Paperno, G. Kruszewski, A. Lazaridou, Q. N. Pham, R. Bernardi, S. Pezzelle, M. Baroni, G. Boleda, and R. Fernández. The lambada dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*, 2016.
- A. Rodriguez, E. Segal, E. Meiri, E. Fomenko, Y. J. Kim, H. Shen, and B. Ziv. Lower numerical precision deep learning inference and training. *Intel White Paper*, 3(1):19, 2018.
- B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.
- H. Shen, N. Mellemudi, X. He, Q. Gao, C. Wang, and M. Wang. Efficient post-training quantization with fp8 formats, 2023.

- X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. *Advances in neural information processing systems*, 32, 2019.
- H. Touvron, T. Lavigra, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.
- H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.
- V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.
- A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- X. Wei, Y. Zhang, Y. Li, X. Zhang, R. Gong, J. Guo, and X. Liu. Outlier suppression+: Accurate quantization of large language models by equivalent and optimal shifting and scaling. *arXiv preprint arXiv:2304.09145*, 2023.
- X. Wu, Z. Yao, and Y. He. Zeroquant-fp: A leap forward in llms post-training w4a8 quantization using floating-point formats. *arXiv preprint arXiv:2307.09782*, 2023.
- G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.
- R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.
- S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

# Fast Distributed Inference Serving for Large Language Models

Bingyang Wu\* Yinmin Zhong\* Zili Zhang\* Gang Huang Xuanzhe Liu Xin Jin

*Peking University*

## Abstract

Large language models (LLMs) power a new generation of interactive AI applications exemplified by ChatGPT. The interactive nature of these applications demand low job completion time (JCT) for model inference. Existing LLM serving systems use run-to-completion processing for inference jobs, which suffers from head-of-line blocking and long JCT.

We present FastServe, a distributed inference serving system for LLMs. FastServe exploits the autoregressive pattern of LLM inference to enable preemption at the granularity of each output token. FastServe uses preemptive scheduling to minimize JCT with a novel skip-join Multi-Level Feedback Queue scheduler. Based on the new *semi* information-agnostic setting of LLM inference, the scheduler leverages the input length information to assign an appropriate initial queue for each arrival job to join. The higher priority queues than the joined queue are skipped to reduce demotions. We design an efficient GPU memory management mechanism that proactively offloads and uploads intermediate states between GPU memory and host memory for LLM inference. We build a system prototype of FastServe based on NVIDIA FasterTransformer. Experimental results show that compared to the state-of-the-art solution Orca, FastServe improves the average and tail JCT by up to 5.1 $\times$  and 6.4 $\times$ , respectively.

## 1 Introduction

Advancements in large language models (LLMs) open new possibilities in a wide variety of areas and trigger a new generation of interactive AI applications. The most notable one is ChatGPT [1] that enables users to interact with an AI agent in a conversational way to solve tasks ranging from language translation to software engineering. The impressive capability of ChatGPT makes it one of the fastest growing applications in history [3]. Many organizations follow the trend to release LLMs and ChatGPT-like applications, such as the New Bing from Microsoft [5], Bard from Google [2], LLaMa from Meta [52], Alpaca from Stanford [51], Dolly from Databricks [4], Vicuna from UC Berkeley [14], etc.

Inference serving is critical to interactive AI applications based on LLMs. The interactive nature of these applications demand low job completion time (JCT) for LLM inference,

in order to provide engaging user experience. For example, users expect their inputs to ChatGPT to be responded instantly. Yet, the size and complexity of LLMs put tremendous pressure on the inference serving infrastructure. Enterprises provision expensive clusters that consist of accelerators like GPUs and TPUs to process LLM inference jobs.

LLM inference has its own unique characteristics (§2) that are different from other deep neural network (DNN) model inference like ResNet [31]. DNN inference jobs are typically deterministic and highly-predictable [29], i.e., the execution time of an inference job is mainly decided by the model and the hardware. For example, different input images have similar execution time on the same ResNet model on a given GPU. In contrast, LLM inference jobs have a special *autoregressive* pattern. An LLM inference job contains multiple iterations. Each iteration generates one output token, and each output token is appended to the input to generate the next output token in the next iteration. The execution time depends on both the input length and the output length, the latter of which is not known *a priori*.

Existing inference serving solutions like Clockwork [29] and Shepherd [59] are mainly designed for deterministic model inference jobs like ResNet [31]. They rely on accurate execution time profiling to make scheduling decisions, which do not work for LLM inference that has variable execution time. Orca [58] is the state-of-the-art solution for LLM inference. It proposes iteration-level scheduling where at the end of each iteration, it can add new jobs to or remove finished jobs from the current processing batch. However, it uses first-come-first-served (FCFS) to process inference jobs. Once a job is scheduled, it runs until it finishes. Because the GPU memory capacity is limited and inference jobs require low JCT, the current processing batch cannot be expanded with an arbitrary number of incoming jobs. It is known that run-to-completion processing has head-of-line blocking [35]. The problem is particularly acute for LLM inference jobs, because the large size of LLMs induces long absolute execution time. A large LLM inference job, i.e., with long output length, would run for a long time to block following short jobs.

We present FastServe, a distributed inference serving system for LLMs. FastServe exploits the autoregressive pattern of LLM inference and iteration-level scheduling to enable preemption at the granularity of each output token. Specifically, when one scheduled job finishes generating an output token, FastServe can decide whether to continue this job or

\*Equal contribution.

preempt it with another job in the queue. This allows FastServe to use preemptive scheduling to eliminate head-of-line blocking and minimize JCT.

The core of FastServe is a novel skip-join Multi-Level Feedback Queue (MLFQ) scheduler. MLFQ is a classic approach to minimize average JCT in information-agnostic settings [8]. Each job first enters the highest priority queue, and is demoted to the next priority queue if it does not finish after a threshold. The key difference between LLM inference and the classic setting is that LLM inference is *semi* information-agnostic, i.e., while the output length is not known *a priori*, the input length is known. Because of the autoregressive pattern of LLM inference, the input length decides the execution time to generate the first output token, which can be significantly larger than those of the later tokens (§4.1). For a long input and a short output, the execution time of the first output token dominates the entire job. We leverage this characteristic to extend the classic MLFQ with skip-join. Instead of always entering the highest priority queue, each arrival job joins an appropriate queue by comparing its execution time of the first output token with the demotion thresholds of the queues. The higher priority queues than the joined queue are skipped to reduce demotions.

Preemptive scheduling with MLFQ introduces extra memory overhead to maintain intermediate state for started but unfinished jobs. LLMs maintain a key-value cache for each Transformer layer to store intermediate state (§2.2). In FCFS, the cache only needs to store the intermediate state of the scheduled jobs in the processing batch, limited by the maximum batch size. But in MLFQ, more jobs may have started but are demoted to lower priority queues. The cache has to maintain the intermediate state for all started but unfinished jobs in MLFQ. The cache can overflow, given the large size of LLMs and the limited memory capacity of GPUs. Naively, the scheduler can pause starting new jobs when the cache is full, but this again introduces head-of-line blocking. Instead, we design an efficient GPU memory management mechanism that proactively offloads the state of the jobs in low-priority queues to the host memory when the cache is close to full, and uploads the state back when these jobs are to be scheduled. We use pipelining and asynchronous memory operations to improve the efficiency.

For large models that do not fit in one GPU, FastServe leverages parallelization strategies including tensor parallelism [50] and pipeline parallelism [33] to perform distributed inference serving with multiple GPUs (§4.3). The scheduler runs multiple batches of jobs concurrently in a pipeline to minimize pipeline bubbles. The key-value cache manager partitions the key-value cache over multiple GPUs to organize a distributed key-value cache, and handles swapping between GPU memory and host memory in a distributed manner.

We implement a system prototype of FastServe based on NVIDIA FasterTransformer [18]. We evaluate FastServe on

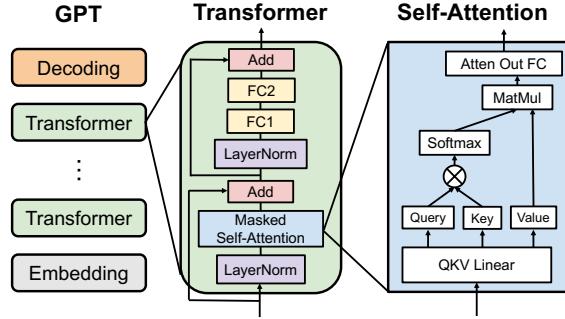
different configurations of GPT models with a range of workloads with varying job arrival rate, burstiness and size. In particular, we evaluate the end-to-end performance of FastServe for GPT-3 175B (the largest GPT-3 model) on 16 NVIDIA A100 GPUs. We also evaluate the design choices and scalability of FastServe. The experiments show that compared to the state-of-the-art solution Orca, FastServe improves the average and tail JCT by up to 5.1 $\times$  and 6.4 $\times$ , respectively.

## 2 Background and Motivation

### 2.1 GPT Inference and Applications

**GPT inference.** GPT [12] is a family of language models based on Transformer [53]. The inference procedure of GPT follows an autoregressive pattern. The input is a sequence of tokens, which is often called a prompt. GPT processes the prompt and outputs the probability distribution of the next token to sample from. We call the procedure of processing and sampling for one output token as an *iteration*. After the model is trained with a large corpus, it is able to accomplish language tasks with high quality. For example, when fed with the input “knowledge is”, it is expected to output a higher probability for “power” than “apple”. After the first iteration, the generated token is appended to the initial prompt and fed into GPT as a whole to generate the next token. This generation procedure will continue until a unique <EOS> token is generated which represents the end of the sequence or a pre-defined maximum output length is reached. This inference procedure is quite different from other models like ResNet, of which the execution time is typically deterministic and highly predictable [29]. Here although the execution of each iteration still holds such properties, the number of iterations (i.e., the output length) is unknown, making the total execution time of one inference job unpredictable.

**GPT applications.** Although GPT is nothing but a language model to predict the next token, downstream NLP tasks can be recast as a generation task with prompt engineering. Specifically, one can append the original input after the text description of the specific task as the prompt to GPT, and GPT can solve the task in its generated output. ChatGPT is a representative application. After supervised fine-tuning for the conversational task and an alignment procedure using Reinforcement Learning from Human Feedback (RLHF) on the original GPT model [1], ChatGPT enables users to interact with an AI agent in a conversational way to solve tasks ranging from translation, question-answering, and summarization to more nuanced tasks like sentiment analysis, creative writing, and domain-specific problem-solving. Despite its power, the interactive nature of ChatGPT imposes tremendous pressure on the underlying inference serving infrastructure. Many users may send jobs to ChatGPT concurrently and expect responses as soon as possible. Therefore, JCT is critical for ChatGPT-like interactive applications.



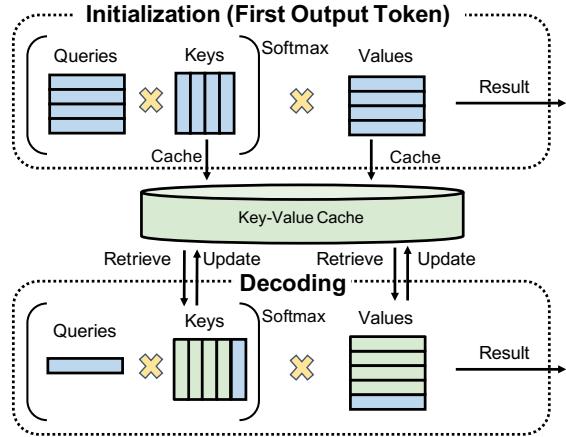
**Figure 1.** GPT model architecture.

## 2.2 Inference Serving Systems

Most existing inference serving systems, such as Tensorflow Serving [43] and Triton Inference Server [19], are agnostic to DNN models. They serve as an abstraction above the underlying execution engine to queue the arriving jobs, dispatch jobs to available computing resources, and return the results to clients. Since accelerators like GPUs have massive amounts of parallel computing units, they typically batch jobs to increase hardware utilization and system throughput. With batching enabled, the input tensors from multiple jobs are concatenated together and fed into the model as a whole. The drawback of batching is higher memory overhead compared to single-job execution. Since the activation memory grows proportionally to model size, the large size of LLMs limits the maximum batch size of LLM inference.

As the popularity of GPT models grows, inference serving systems have evolved to include optimizations specific to the unique architecture and iterative generation pattern of GPT. The major part in GPT’s architecture is a stack of Transformer layers, as shown in Figure 1. In a Transformer layer, the Masked Self-Attention module is the core component that distinguishes it from other architectures like CNNs. For each token in the input, it derives three values, which are query, key, and value. It takes the dot products of query with all the keys of previous tokens to measure the interest of previous tokens from the current token’s point of view. Since GPT is a language model trained to predict the next token, each token should not see information after its location. This is implemented by causal masking in Transformer. It then applies the Softmax to the dot products to get weights and produces the output as a weighted sum of the values according to the weights. At a high level, the attention operator makes each token in the input aware of other tokens regardless of the location distance.

During each iteration of GPT inference, for each token, the attention operator requires the keys and values of its preceding tokens. A naive, stateless implementation always recomputes all the keys and values in each iteration. To avoid such recomputation overhead, fairseq [44] suggests saving the keys and values in a *key-value cache* across iterations for reuse. In this way, the inference procedure can be divided into



**Figure 2.** Example usage of KV cache in two phases. The prompt contains four tokens. Blue grids represent newly computed tensors in its iteration, while green grids represent reused tensors from the key-value cache.

two phases. Figure 2 illustrates the usage of key-value cache in different phases. In the *initialization phase*, i.e., the first iteration, the prompt is processed to generate the key-value cache for each transformer layer of GPT. In the *decoding phase*, GPT only needs to compute the query, key, and value of the newly generated token. The key-value cache is utilized and updated to generate tokens step by step. Thus the execution time of iterations in the decoder phase is usually smaller than that of the first iteration. Other system libraries optimized for Transformers such as HuggingFace [56] and FasterTransformer [18] also perform the same optimization.

Another optimization is iteration-level scheduling proposed by Orca [58]. Naive job-level scheduling executes a batch of jobs until all jobs finish. The jobs that finish early cannot return to clients, while newly arrived jobs have to wait until the current batch finishes. Instead, the iteration-level scheduling invokes the execution engine to run only a single iteration on the batch each time, i.e., generate one output token for each job. After each iteration, the finished jobs can leave the batch, and the arrived jobs can join the batch. However, the maximum batch size is limited by the GPU memory capacity, and the low-latency requirement of interactive applications also affects the choice of batch size.

## 2.3 Opportunities and Challenges

**Opportunity: preemptive scheduling.** The major limitation of existing inference serving systems for LLMs [18, 58] is that they use simple FCFS scheduling and run-to-completion execution, which has head-of-line blocking and affects JCT. Head-of-line blocking can be addressed by preemptive scheduling. For LLM inference, each job consists of multiple iterations, and each iteration generates one output token. The opportunity is to leverage this autoregressive pattern to enable preemptions at the granularity of each iteration, i.e.,

preempting one job when it finishes generating an output token for another job. With the capability of preemption, the scheduler can use preemptive scheduling policies to avoid head-of-line blocking and optimize for JCT.

**Challenge 1: unknown job size.** Shortest Remaining Processing Time (SRPT) [47] is a well-known preemptive scheduling policy for minimizing average JCT. However, SRPT requires knowledge of the remaining job size. Different from one-shot prediction tasks such as image classification, LLM inference is iterative. While the execution time of one iteration (i.e., generating one output token) can be profiled based on the model architecture and the hardware, the number of iterations (i.e., the output sequence length) is unknown and is also hard to predict, because it is determined by the semantics of the job. Therefore, SRPT cannot be directly applied to LLM inference to minimize average JCT.

**Challenge 2: GPU memory overhead.** Preemptive scheduling policies introduce extra GPU memory overhead for LLM inference. FCFS with run-to-completion only needs to maintain the key-value cache for the ongoing jobs. In comparison, preemptive scheduling has to keep the key-value cache in the GPU memory for all preempted jobs in the pending state for future token generation. The key-value cache consumes a huge amount of GPU memory. For example, the key-value cache for a single job of GPT-3 175B with input sequence length = 512, requires at least 2.3GB memory (§4.2). The GPU memory capacity limits the key-value cache size and affects the preemptive scheduling policies.

## 3 FastServe Overview

### 3.1 Desired Properties

As LLM applications like ChatGPT are becoming popular, delivering high-performance LLM inference is increasingly important. LLMs have their own characteristics that introduce challenges to distributed computation and memory consumption. Our goal is to build an inference serving system for LLMs that meet the following three requirements.

- **Low job completion time.** We focus on interactive LLM applications. Users expect their jobs to finish quickly. The system should achieve low job completion time for processing inference jobs.
- **Efficient GPU memory management.** The model parameters and KV cache of LLMs consume tremendous GPU memory. The system should efficiently manage GPU memory to store the model and intermediate state.
- **Scalable distributed execution.** LLMs require multiple GPUs to perform inference in a distributed manner. The system should provide scalable distributed execution to process LLM inference jobs.

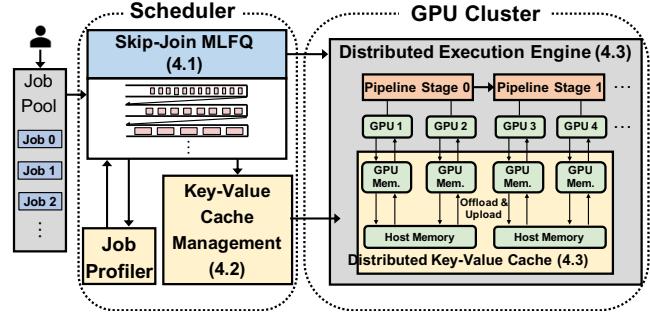


Figure 3. FastServe architecture.

### 3.2 Overall Architecture

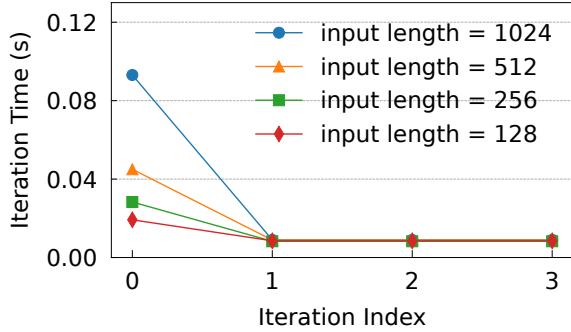
Figure 3 illustrates the architecture of FastServe. Users submit their jobs to the job pool. The skip-join MLFQ scheduler (§4.1) utilizes a profiler to decide the initial priority of newly arrived jobs based on their initiation phase execution time. It adopts iteration-level preemption and favors the least-attained job to address head-of-line blocking issue. Once a job is chosen to be executed, the scheduler sends it to the distributed execution engine (§4.3) which serves the LLM in a GPU cluster and interacts with the distributed key-value cache to retrieve and update the key-value tensors for the corresponding job during runtime. To address the problem of limited GPU memory capacity, the key-value cache manager (§4.2) proactively offloads the key-value tensors of the jobs with low priority to the host memory and dynamically adjusts its offloading strategy based on the burstiness of workload. To scale the system to serve large models like GPT-3 175B, FastServe distributes the model inference across multiple GPUs. Extensions are added to the scheduler and key-value cache to support distributed execution.

## 4 FastServe Design

In this section, we first describe the skip-join MLFQ scheduler to minimize JCT. Then we present the proactive KV cache management mechanism to handle the GPU memory capacity constraint. At last, we show how to apply these techniques to the distributed setting.

### 4.1 Skip-Join MLFQ Scheduler

**Strawman: naive MLFQ.** Because the job size of LLM inference is unknown, SRPT cannot be directly applied. Least-attained service (LAS) is known to approximate SRPT in information-agnostic settings, and MLFQ is a practical approach that realizes discretized LAS to reduce job switching and has been used in many scheduling systems [6, 8, 15, 28, 32]. MLFQ has a number of queues, each assigned with a different priority level. An arrival job first enters the highest priority queue and is demoted to the next level queue if it does not finish after a *demotion threshold*, i.e., quantum, which is a tunable parameter assigned to each queue. Higher priority queues usually have a shorter quantum.

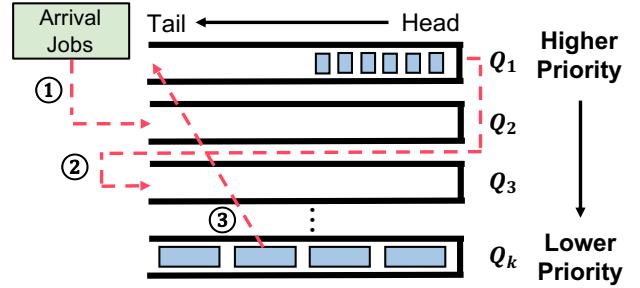


**Figure 4.** The execution time of the first four iterations (i.e., first four output tokens) with different input sequence length.

Although MLFQ assumes no prior knowledge of the job size, it is not well suited for LLM serving. Specifically, the first iteration time of a job with a long input sequence length may exceed the quantum of the highest priority queue. When the job gets scheduled, it would use up the quantum in the middle of its first iteration. This creates a dilemma for scheduling. If the scheduler preempts the job, the intermediate activations have to drop and recompute later, which wastes computing resources and time. If the scheduler does not preempt it, then the scheduler violates the design purpose of MLFQ and suffers from head-of-line blocking again.

**Our solution: skip-join MLFQ.** Our setting differs from the classic information-agnostic setting in that LLM inference is *semi* information-agnostic setting. We leverage the characteristics of LLM inference to address the problem of the naive MLFQ. Specifically, although the number of iterations (i.e., the output length) is not known ahead of time, the execution time of each iteration is predictable. The iteration time is determined by a few key parameters such as the hardware, the model, and the input length, and thus can be accurately profiled in advance. Figure 4 shows the iteration time for GPT-3 2.7B on NVIDIA A100 under different input sequence length. We can see that the first iteration time (i.e., the execution time to generate the first output token) is longer than those in the decoding phase within a single job. As the input sequence length increases, the first iteration time grows roughly in a linear manner, while the increase of the iteration time in the decoding phase is negligible. This is due to the key-value cache optimization (§2.2). In the first iteration, all the key-value tensors of the input tokens are computed and cached. While in the following iterations, only the key-value tensors of the newly generated token require computation and others are loaded from the key-value cache, changing the bottleneck from computing to memory bandwidth.

Based on these observations, we design a novel skip-join MLFQ scheduler for LLM inference. Figure 5 highlights the core scheduling operations, and Algorithm 1 shows the pseudo-code. The scheduler uses the basic MLFQ framework with a skip-join feature for new jobs. The quantum of  $Q_1$  is set



**Figure 5.** Skip-join MLFQ with starvation prevention.

to the minimum iteration time and the ratio between  $Q_i$  and  $Q_{i-1}$  is controlled by a parameter *quantum ratio*. We set it to 2 by default and our experiments (§6.3) show that FastServe’s performance is not sensitive to this quantum setting. After finishing an iteration for the jobs in current processing batch, the scheduler preempts these jobs  $J_{pre}$  and invokes the procedure *SkipJoinMLFQScheduler*. This procedure handles newly arrived job  $J_{in}$  and constructs a new batch of jobs  $J_{out}$  for the execution of next iteration.

The scheduler accurately assigns priority to a newly arrived job based on its first iteration time, which is determined by the input sequence length. Specifically, when a job arrives, its priority is set to the highest priority whose quantum is larger than the job’s first iteration time using the *getHighestPriority* method (lines 7-8). Then the scheduler ① skip-joins the job into its corresponding queue rather than the highest priority queue in the naive MLFQ (line 9). For preempted jobs, the scheduler returns the newly generated tokens to the clients immediately, rather than returning the entire response until the completion of the job, which optimizes the user experience (line 12). If the job does not finish and uses up its quantum in the current queue, the scheduler decides the demoted priority of the job based on its current priority and next iteration time by using *getDemotionPriority* and ② demotes it to the corresponding queue (lines 17-20). The skip-join and demotion operations may cause the jobs with long input length or output length to suffer from starvation. To avoid this, the scheduler periodically resets the priority of a job and ③ promotes it to the highest priority queue  $Q_1$ , if it has been in the waiting state longer than a promotion threshold, *STARVE\_LIMIT* (lines 22-26). The promoted job will get an extra quantum if its next iteration time is less than the quantum of  $Q_1$  to ensure its next iteration without preemption. This creates possibility of head-of-line blocking, so the system administrator of FastServe can tune *STARVE\_LIMIT* to make a tradeoff between performance and starvation. At last, the scheduler selects a set of jobs with the highest priority without exceeding the maximum batch size, which constrained by the GPU memory capacity (lines 28-31). By utilizing the characteristics of LLM inference, the skip-join MLFQ scheduler can adjust the

---

**Algorithm 1** Skip-Join Multi-Level Feedback Queue Scheduler

---

```

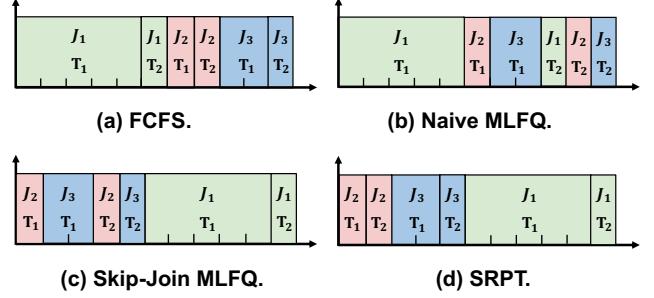
1: Input: Queues  $Q_1, Q_2, \dots, Q_k$ , newly arrived jobs  $J_{in}$ , pre-
   empted jobs  $J_{pre}$ , and profiling information  $P$ 
2: Output: Jobs to be executed  $J_{out}$ 
3: procedure SKIPJOINMLFQSCHEDULER
4:   Initialization:  $J_{out} \leftarrow \emptyset$ .
5:   // Process newly arrival jobs.
6:   for  $job \in J_{in}$  do
7:      $nextIterTime \leftarrow P.getNextIterTime(job)$ 
8:      $p_{job} \leftarrow getHighestPriority(nextIterTime)$ 
9:      $Q_{p_{job}}.push(job)$ 
10:  // Process preempted jobs.
11:  for  $job \in J_{pre}$  do
12:     $job.outputNewGeneratedToken()$ 
13:     $p_{job} \leftarrow job.getCurrentPriority()$ 
14:    if  $job.isFinished()$  then
15:       $Q_{p_{job}}.pop(job)$ 
16:      continue
17:    if  $job.needDemotion()$  then
18:       $nextIterTime' \leftarrow P.getNextIterTime(job)$ 
19:       $p'_{job} \leftarrow getDemotionPriority(p_{job}, nextIterTime')$ 
20:       $r.demoteTo(Q_{p'_{job}})$ 
21:    // Promote starved jobs.
22:    for  $q \in \{Q_2, Q_3, \dots, Q_k\}$  do
23:      for  $job \in q$  do
24:        if  $job.needPromotion()$  then
25:           $job.promoteTo(Q_1)$ 
26:           $job.resetStarveTimer()$ 
27:    // Schedule jobs to be executed.
28:    for  $q \in \{Q_1, Q_2, \dots, Q_k\}$  do
29:      for  $job \in q$  do
30:        if  $job.isReady() \text{ and } |J_{out}| < MaxBatchSize$  then
31:           $J_{out}.push(job)$ 

```

---

job priority more accurately and reduce demotions. Thus it achieves better approximation to SRPT than the naive MLFQ.

**Example.** Figure 6 shows an example to illustrate our scheduler and compares it against the alternatives. Three jobs arrive at time 0 in the order of  $J_1, J_2, J_3$ , where their first iteration times are 5, 1, and 2, respectively, and their output lengths are all equal to 2. We assume the iteration time in the decoding phase is 1 for simplicity. Skip-join MLFQ and Naive MLFQ both have four priority queues with quantum 1, 2, 4, and 8. For Naive MLFQ, it does interrupt the iteration if a job uses up its quantum during execution. The average JCT of FCFS, naive MLFQ, skip-join MLFQ, and SRPT are 8.33, 10, 6.67, and 6, respectively. In general, the algorithms with more information perform better than those with less information in minimizing JCT. Without skip-join, naive MLFQ may degenerate to round-robin and be worse than FCFS in some cases.



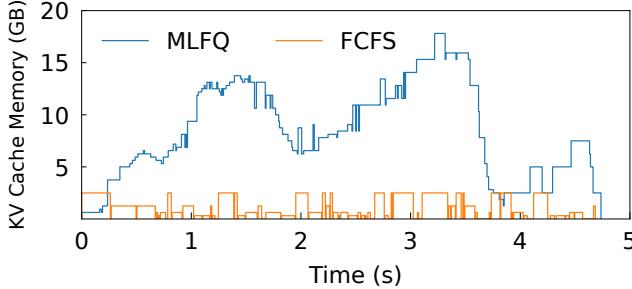
**Figure 6.** Execution timeline of three jobs with different scheduling algorithms. The job index  $J_i$  and generated token index  $T_i$  are marked in each iteration block.

## 4.2 Proactive Key-Value Cache Management

The skip-join MLFQ scheduler provides iteration-level pre-emption to approximate SRPT without knowing the exact job size. However, preemption also increases the number of ongoing jobs in the system, which introduces extra GPU memory overhead. Formally, for a particular LLM inference serving job, denote the input sequence length by  $s$ , the output sequence length by  $t$ , the hidden dimension of the transformer by  $h$ , and the number of transformer layers by  $l$ . If the model weights and all computations are in FP16, the total number of bytes to store the key-value cache for this single job is  $4 \times lh(s + t)$ . Take GPT-3 175B as an example ( $l = 96, h = 12288$ ). Given an input sequence length  $s = 512$  and a minimum output sequence length  $t = 1$ , the GPU memory overhead for a single job is as high as 2.3GB. As the generation continues, its output sequence length  $t$  will increase, which further increases the GPU memory overhead.

The schedulers using the run-to-completion policy can tolerate this memory overhead because the maximum number of ongoing jobs would not exceed the size of the current processing batch. Figure 7 shows the key-value cache memory consumption of FCFS and skip-join MLFQ for GPT-3 2.7B model under a synthetic workload. Although we choose a relatively small model and limit the maximum output length to 20, the peak KV cache memory overhead for skip-join MLFQ can be 7 $\times$  larger than that of FCFS. In a more realistic scenario where the model size scales to 175B and the output length can be more than a thousand, the memory overhead for skip-join MLFQ can easily exceed the memory capacity of NVIDIA’s newest Hopper 80 GB GPUs.

**Strawman solution 1: defer newly arrived jobs.** A naive solution is to simply *defer* the execution of newly arrived jobs when the GPU memory is not sufficient to hold additional key-value tensors and keep scheduling current jobs until they finish. Although new jobs often have higher priority, they have to be blocked to wait for free memory space. Under extreme GPU memory-constrained settings, this solution



**Figure 7.** The key-value cache memory consumption for GPT-3 2.7B under different schedulers. The workload follows a Gamma Process with rate=64 and CV=4. The maximum output length is set to 20 to avoid GPU out of memory.

would degenerate MLFQ to FCFS, which again suffers from head-of-line blocking.

**Strawman solution 2: kill low-priority jobs.** Another straightforward solution is to *kill* some low-priority jobs and free their key-value cache to make room for newly arrived high-priority jobs. This solution has two problems. First, the killed jobs lose their generation state and need to rerun the initiation phase to generate their key-value cache, which wastes computation resources and time. Second, it may cause deadlocks. When the high-priority jobs keep arriving, ongoing jobs with lower priority would be killed. With the starvation avoidance mechanism enabled, the killed jobs may be promoted to the highest-priority queue after *STARVE\_LIMIT*. In this case, the promoted job may again kill the currently executing job that kills it in the previous round, which leads to a deadlock. This brings extra complexity to set *STARVE\_LIMIT*. A large value causes starvation, while a small value may cause deadlocks.

**Our solution: proactive key-value cache swapping.** From the two strawman solutions, we can see a dilemma that MLFQ requires more GPU memory for better performance, while the limited GPU memory restricts the potential of the scheduling based on MLFQ. To solve this problem, our key observation is that the key-value tensors only need to be reserved in the GPU memory when its corresponding job gets scheduled. Based on this observation, FastServe can offload inactive key-value tensors of jobs to the host memory and upload necessary key-value tensors back to the GPU memory when they are needed. The challenge is that the overhead of offloading and uploading is not negligible compared to the token generation time. When deploying GPT-3 175B on NVIDIA A100 GPUs, the key-value tensors of a job can occupy 2.3 GB memory. The token generation time in decoding phase is about 250ms, while the time to transfer the key-value tensors between host memory and GPU memory with PCIe 4.0×16 full bandwidth is about 36ms.

FastServe uses *proactive* offloading and uploading to minimize the swapping overhead. Instead of reactively offloading

jobs when the key-value cache is full, FastServe keeps some idle key-value cache slot for newly arrived jobs. When a new job arrives, it can get a key-value cache slot immediately without incurring the overhead of offloading a preempted job. Rather than reactively uploading the key-value tensors for the executed job, when the key-value cache space on the GPU is sufficient, FastServe proactively uploads the key-value tensors of the jobs that will be used in the near future so that the token generation can be overlapped with the data transmission.

The number of idle key-value cache is the maximum of a tunable parameter set by the system administrator,  $K$ , and the value provided by a burst predictor. The tunable parameter  $K$  ensures that at least  $K$  newly arrived job will not be blocked by the offloading. The burst predictor is a heuristic that predicts the number of jobs that will arrive in the near future. When a burst of jobs arrives, the predictor leaves more idle key-value cache slots in advance. We use the number of jobs in the top  $K'$  priority queues as the prediction, where  $K'$  is also a tunable parameter. Empirically, we find that the performance is not sensitive to the choices of  $K$  and  $K'$ .

**Job swapping order.** To mitigate the impact of job swapping, the decision on the order of offloading and uploading is made based on a metric, the estimated next scheduled time (ENST). The ENST is the time when the job will be scheduled to execute next time. The job with the largest ENST will be offloaded first, and the job with the smallest ENST will be uploaded first. In general, the lower priority a job has, the later it will be scheduled to execute. However, due to the starvation prevention mechanism, a job with a lower priority may be promoted to a higher priority queue. In this case, a job with a low priority may also be executed first.

To handle this case, for job  $i$ , FastServe considers the time to promote this job and the sum of executed time of all jobs with higher priorities before executing  $i$ . Formally, let the time to promote as  $T_{\text{promote}}(i)$ . As for the sum of executed time of all jobs with higher priorities before executing  $i$ , we assume those jobs do not finish earlier before being demoted to the priority queue of  $i$ . In this case, the execution time of job  $j$  with a higher priority can be calculated as follows:

$$T_{\text{execute}}(i, j) = \sum_{i.\text{priority} < k \leq j.\text{priority}} \text{quantum}(k)$$

where  $i.\text{priority}$  is the priority of job  $i$ , and  $\text{quantum}(k)$  is the quantum of the priority queue with priority  $k$ . Based on this, the sum of executed time of all jobs with higher priorities than job  $i$  is defined as:

$$T_{\text{execute}}(i) = \sum_{i.\text{priority} < j.\text{priority}} T_{\text{execute}}(i, j)$$

At last, taking both the promotion for starvation prevention and the execution of higher priority jobs into consideration,

the ENST of job  $i$  is calculated as:

$$ENST(i) = \min(T_{promote}(i), T_{execute}(i))$$

This ENST definition estimates how long job  $i$  will be scheduled to execute. Therefore, using this metric to decide the order of offloading and uploading makes the key-value tensors of active jobs more likely on the GPU memory, and those of inactive jobs more likely on the host memory. This hides the swapping overhead as much as possible.

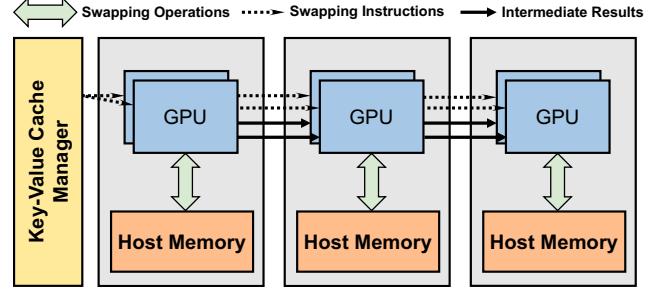
### 4.3 Support for Distributed LLM Serving

Previous research shows that the capability of LLMs empirically conforms to the scaling law in terms of the number of model parameters [37]. The more parameters an LLM has, the more powerful an LLM can be. However, the memory usage of an LLM is also proportional to the number of parameters. For example, GPT-3 175B when stored in half-precision, occupies 350GB GPU memory to just hold the weights and more for the intermediate state during runtime. Therefore, LLM often needs to be split into multiple pieces and served in a distributed manner with multiple GPUs.

Tensor parallelism [42, 50] and pipeline parallelism [33, 41] are two most widely-used techniques for distributed execution of deep learning models. FastServe supports the hybrid of these two parallel techniques for serving LLMs. An LLM is composed of a series of operators over multi-dimensional tensors. Tensor parallelism splits each operator across multiple devices, with each device executing a portion of the computation in parallel. Additional communication overhead is required to split the input and collect the output from participating GPUs. Tensor parallelism expands the computation and memory available to a single job, thus reduces the execution time for each iteration.

Pipeline parallelism splits the operators of an LLM computation graph into multiple stages and executes them on different devices in a pipeline fashion. During inference, each stage computes part of the entire computation graph and transmits the intermediate results to the next stage in parallel. Pipeline parallelism requires less communication overhead compared to tensor parallelism and also allows the LLM to exceed the memory limitation of a single GPU. Since multiple processing batches are under processing simultaneously in different stages, FastServe needs to handle multiple batches in the distributed engine at the same time.

**Job scheduling in distributed serving.** In the traditional MLFQ setting, if no new job arrives, the scheduler would schedule the job with the highest priority and executes it until it finishes or is demoted. However, when using pipeline parallelism, the scheduler schedules at the granularity of the stage. When a job finishes the first stage and sends the intermediate result to the next stage, the scheduler needs to decide on the next job to execute. In this case, the scheduler cannot follow the traditional MLFQ that keeps scheduling



**Figure 8.** Overlapping key-value cache offloading with intermediate result transmission to reduce offloading overhead.

the same job until demotion, because the job is still running. To preserve the semantics of MLFQ, FastServe still keeps the running job in the priority queue, and each time selects the highest priority job in the pending state to execute. Therefore, the early job in a queue can finish the quantum more quickly.

**Key-value cache management in distributed serving.** Because the key-value cache occupies a large fraction of GPU memory, the key-value cache of FastServe is also partitioned across multiple GPUs in distributed serving. In LLM inference, each key-value tensor is used by the same stage of the LLM. Therefore, FastServe partitions key-value tensors as tensor parallelism requires, and assigns each key-value tensor to the corresponding GPU so that all computation on a GPU only needs local key-value tensors.

The proactive key-value cache swapping mechanism of FastServe is also distributed. Because different stages of the LLM process different jobs at the same time, each stage may offload or upload different key-value tensors independently. To reduce redundant control, before processing the intermediate result sent from the previous stage, the current stage does the same offloading or uploading action as the previous stage does. The intermediate result transmission and key-value cache swapping occur in parallel, so the overhead of key-value cache swapping is further reduced. As shown in Figure 8, when the intermediate result is sent to the next stage, the next stage receives the swapping instructions and can swap the key-value cache at the same time if needed. The key-value cache swapping mechanism only needs to decide the offloading or uploading of the first stage. When using tensor parallelism splitting the first stage into multiple chunks, a centralized key-value cache swapping manager instructs all chunks in the first stage to offload or upload the key-value tensors owned by the same job.

## 5 Implementation

We implement FastServe with 10,000 lines of code in Python and C++. The distributed execution engine is based on NVIDIA FasterTransformer[18] which is a high-performance transformer library with custom CUDA kernel implementation.

We modify it to support iteration-level scheduling and interact with the key-value cache manager. We also add extensions to its pipeline parallelism because its original job-level scheduling implementation does not allow injecting another batch before the finish of the currently running batch. It can only split a batch of jobs into multiple microbatches [33] and pipelines the executions of different pipeline stages across the microbatches. This loses the chance to pipeline execution between job batches, and smaller microbatches reduce device utilization. In our implementation, the execution engine can receive a new batch of jobs as soon as the first pipeline stage finishes execution, which means every partition of the model processes one of the batches without being idle.

We implement the key-value cache manager with MPI [26] in a distributed manner, because the key-value tensors are produced and consumed on different GPUs. The distributed design makes it possible to save and retrieve the key-value tensors on the corresponding GPUs, which minimizes the data transfer overhead. We also use MPI to pass messages to synchronize the offloading procedure across the GPUs and utilize multiple CUDA streams to overlap the computation with proactive swapping.

## 6 Evaluation

In this section, we first use end-to-end experiments to demonstrate the overall performance improvements of FastServe over state-of-the-art LLM serving systems on GPT-175B[12]. Next, we deep dive into FastServe to evaluate its design choices and show the effectiveness of each component in FastServe under a variety of settings. Last, we analyze the scalability of FastServe under different numbers of GPUs.

### 6.1 Methodology

**Testbed.** The end-to-end (§6.2) and scalability (§6.4) experiments use two AWS EC2 p4d.24xlarge instances. Each instance is configured with eight NVIDIA A100 40GB GPUs connected over NVLink, 1152 GB host memory, and PCIe 4.0×16. Due to the limited budget, the experiments for design choices (§6.3) use one NVIDIA A100 40GB GPU in our own testbed to validate the effectiveness of each component.

**LLM models.** We choose the representative LLM family, GPT [12], for evaluation, which is widely used in both academics and industry. In LLM serving, the large model weights are usually pre-trained and then fine-tuned into different versions to serve different tasks. We select several widely used model sizes [12] for different experiments. Table 1 lists the detailed model sizes and model configurations. We use FP16 precision for all experiments in our evaluation.

**Workloads.** Similar to prior work on LLM serving [58], we synthesize a trace of jobs to evaluate the performance of FastServe, since there is no publicly-available job trace for LLM inference. The job size is generated by sampling a random input and output length from a Zipf distribution which

Model	Size	# of Layers	# of Heads	Hidden Size
GPT-3 2.7B	5.4GB	32	32	2560
GPT-3 66B	132GB	64	72	9216
GPT-3 175B	350GB	96	96	12288

**Table 1.** Model configurations.

is broadly adopted in many open-source big data benchmarks [13, 17, 27, 55]. The Zipf distribution is parameterized by one parameter,  $\theta$ , which controls the skewness of the distribution. The larger  $\theta$  is, the more skewed the workload is, with more long-tail jobs appearing in the workload. We generate the arrival time for each job following a Gamma process parameterized by arrival rate and coefficient of variation (CV). By scaling the rate and CV, we can control the rate and burstiness of the workload, respectively.

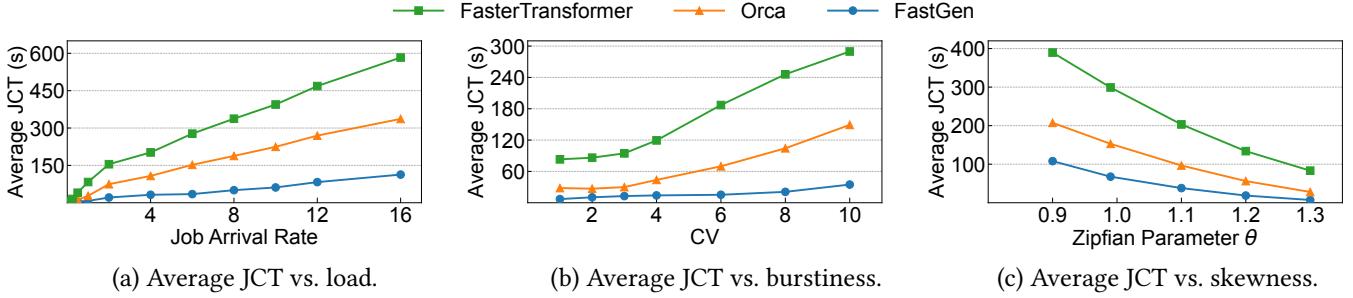
**Metrics.** Since the user-perceived latency is a critical measurement for interactive applications like ChatGPT, which FastServe targets at, we use job completion time (JCT) as the major evaluation metric. Due to limited space, we show average JCT for most experiments, and report both average and tail JCT in the scalability experiments.

**Baselines.** We compare FastServe with two baselines.

- **FasterTransformer** [18]: It is an open-source production-grade distributed inference engine from NVIDIA, which optimizes for large transformer-based language models and is widely used in industry. It supports both tensor parallelism and pipeline parallelism for distributed execution. However, it adopts request-level scheduling and thus does not support pipelining across different jobs as discussed in section §5.
- **Orca** [58]: It is the state-of-the-art LLM serving system that supports iteration-level scheduling and inter-job pipeline parallelism to reduce pipeline bubbles. However, it uses a simple FCFS scheduler with run-to-completion execution, which suffers from head-of-line blocking. Since Orca is not open-sourced, we implement Orca on top of Faster-Transformer for a fair comparison.

### 6.2 Overall Performance

In this subsection, we compare the performance of FastServe to the two baseline systems under a variety of workload settings on GPT-175B. We use two AWS p4d.24xlarge instances with 16 NVIDIA A100 40GB GPUs in total. We use a mix of tensor parallelism and pipeline parallelism. Specifically, the model is partitioned with tensor parallelism in each instance as the eight A100 GPUs in each instance are connected over NVLink with high bandwidth. The two instances execute the jobs through pipeline parallelism which is connected over Ethernet. FastServe significantly outperforms the two baseline systems with its skip-join MLFQ scheduler and proactive key-value cache management, which we summarize as follows.



**Figure 9.** End-to-end performance when serving GPT-3 175B with 16 NVIDIA A100 40GB GPUs.

**Average JCT vs. load.** Figure 9(a) varies the job arrival rate while keeping other parameters (e.g., CV and Zipf parameter) the same. FastServe outperforms Orca by 1×–4.3× and FasterTransformer by 1.9×–11.4×. When the rate is low ( $\leq 0.5$ ), FastServe has the same performance as Orca but outperforms FasterTransformer by around 2×. This is because MLFQ deteriorates into FCFS at a low job arrival rate. FasterTransformer does not support inter-job pipelining which leads to only 50% GPU utilization due to the bubbles in the pipeline. As the rate grows, the head-of-line blocking problem of FCFS becomes more severe. FastServe consistently outperforms Orca by at least 3× and FasterTransformer by at least 5× when the rate is greater than 0.5. FastServe is able to effectively reduce the head-of-line blocking by prioritizing the short jobs with skip-join MLFQ.

**Average JCT vs. burstiness.** Figure 9(b) varies the CV, which controls the burstiness of job arrivals while keeping other parameters (e.g., rate and Zipfian parameter) the same. FastServe outperforms Orca by 2.3×–5.1× and FasterTransformer by 7.4×–12.2×. When the CV is low, the jobs arrive repositively. As a result, the performance gap between FastServe and the two baselines is small. However, when the CV is high, the jobs arrive in a bursty manner, which exacerbates the head-of-line blocking problem. The bursty workload also introduces significant pressure on key-value cache management. With the proactive swapping mechanism, FastServe significantly outperforms the two baselines under high CV.

**Average JCT vs. skewness.** Figure 9(c) varies the Zipfian parameter  $\theta$ , which controls the skewness of the input and output sequence lengths (i.e., the skewness of job size) while keeping other parameters (e.g., rate and CV) the same. FastServe outperforms Orca by 1.9×–3.9× and FasterTransformer by 3.6×–10.6×. When  $\theta$  is small, the input and output lengths of the jobs are more balanced. As a result, the performance gap between FastServe and the two baselines is small. When  $\theta$  becomes large, the input and output lengths of the jobs are more skewed. Thus, FastServe benefits more from the skip-join MLFQ scheduler to tame the head-of-line blocking problem. It is worth noting that the absolute value of JCT

increases as  $\theta$  decreases. This is because we bound the maximum input and output lengths. As a result, the workloads with smaller  $\theta$  (i.e., balanced job lengths) have more tokens to process.

### 6.3 Benefits of Design Choices

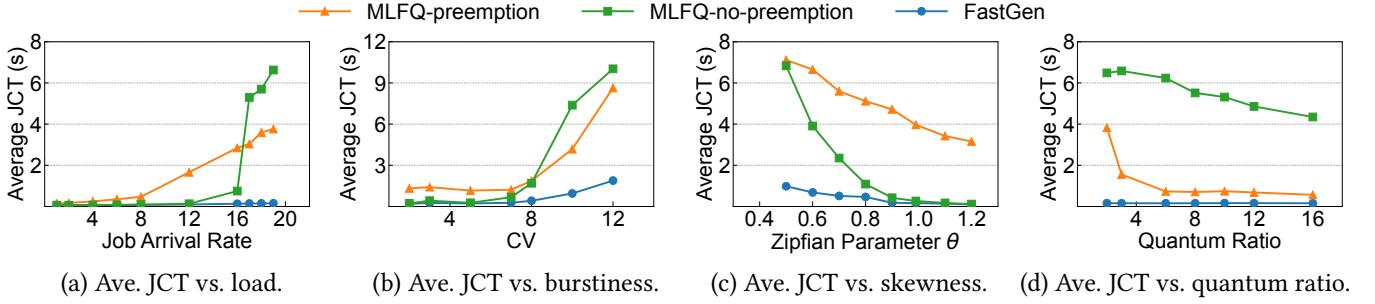
In this subsection, we study the effectiveness of FastServe’s main techniques: skip-join MLFQ scheduler and proactive key-value cache management. Due to a limited budget, we use one A100 GPU to run GPT-3 2.7B in the experiments.

**Benefits of skip-join MLFQ.** To show the benefits of the skip-join MLFQ scheduler, we compare it with two baseline MLFQ schedulers.

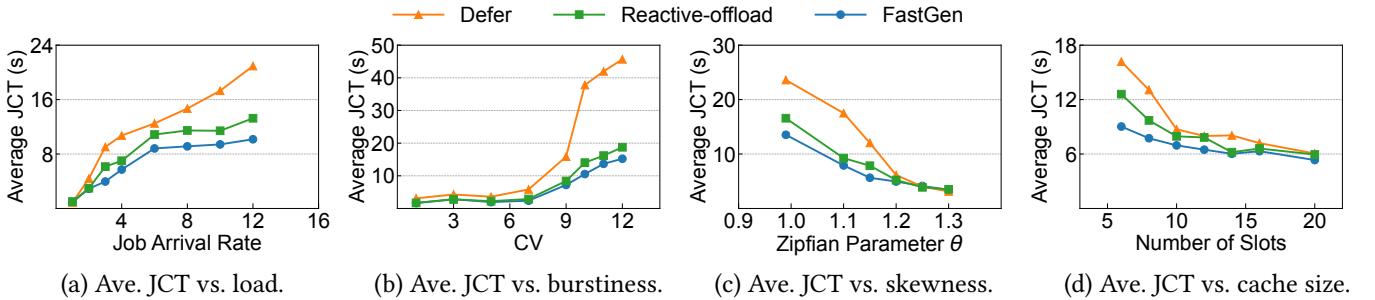
- **MLFQ with preemption (MLFQ-preemption):** It is agnostic to the input length, and puts a newly arrived job to the queue with the highest priority. If the corresponding quantum is not enough to execute an iteration, it preempts (i.e., kills) the current iteration and demotes the job.
- **MLFQ without preemption (MLFQ-no-preemption):** It is also agnostic to the input length. However, if the corresponding quantum is not enough, it continues to execute the halfway iteration and then demotes the job. It degenerates to round-robin scheduling if the quantum is always insufficient.

Similar to previous experiments, we vary the rate, CV, and Zipfian parameter of the workload. In addition, to evaluate the sensitivity of MLFQ to the quantum settings, we vary the *quantum ratio* (§4.1) to see the impact on performance. The results are summarized as follows.

As Figure 10(a) shows, when the rate is low, there is little or even no queueing. Thus, all three schedulers degenerate to FCFS and have similar performance. As the rate grows, MLFQ-preemption suffers from re-execution overhead of halfway iterations since the quantum of the high-priority queues may be not enough to execute the first iteration of some jobs. As for MLFQ-no-preemption, its average JCT increases dramatically when the rate is slightly over 16 since some large jobs in the highest priority queue block too many jobs from execution. As a result, FastServe outperforms the



**Figure 10.** Benefits of the skip-join MLFQ scheduler in FastServe.



**Figure 11.** Benefits of the proactive key-value cache management mechanism in FastServe.

two baseline MLFQ schedulers by up to 24× through its skip-join technique. When varying CV, the performance gap is similar. As shown by Figure 10(b), FastServe outperforms the two baselines by up to 7.8×.

As demonstrated by Figure 10(c), FastServe consistently outperforms MLFQ-preemption and MLFQ-no-preemption by up to 32×, due to the re-execution overhead of MLFQ-preemption and the head-of-line blocking problem of MLFQ-no-preemption. The performance gap between FastServe and MLFQ-no-preemption becomes larger as the Zipfian parameter  $\theta$  decreases. This is because a small  $\theta$  leads to a more balanced distribution of input lengths for each job, making more jobs' first iteration time surpass the quantum of the first few high-priority queues. As a result, MLFQ-no-preemption degenerates to round-robin, so FastServe significantly outperforms MLFQ-no-preemption under such conditions.

In Figure 10(d), it is worth noting that increasing the quantum ratio has little impact on the performance of FastServe, but it reduces the JCT of the two baseline MLFQ schedulers. This demonstrates that FastServe is not sensitive to the quantum settings, making the life of the system administrator much easier. For MLFQ-preemption, enlarging the quantum of each priority queue mitigates the re-execution overhead of preempted inference jobs. For MLFQ-no-preemption, a small quantum makes each job get processed in a round-robin fashion. The problem is mitigated as the quantum increases, so MLFQ-no-preemption performs better. Also, we can see a performance gap between the two baselines even when

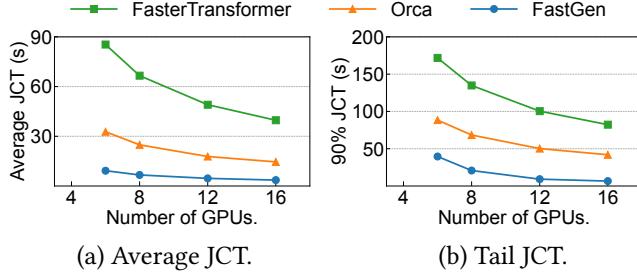
the quantum ratio grows to 16, indicating that compared to re-execution overhead, head-of-line blocking is a more severe performance issue. With skip-join MLFQ, FastServe is able to address the problems of the two baseline MLFQ schedulers and outperforms both of them. Overall, FastServe outperforms the two baseline MLFQ schedulers by 3.6×–41×.

**Benefits of proactive key-value cache management.** To show the benefits of the proactive key-value cache management mechanism, we compare it with two baseline key-value cache management mechanisms.

- **Defer:** It defers an upcoming job if the key-value cache slots are all used. The job waits until a key-value cache slot is available.
- **Reactive-offload:** When the key-value cache is full and a scheduled job is unable to get an empty slot, it reactively picks a job in the cache and offloads its state to the host memory. The cache replacement policy (i.e., picking which job to offload) is the same as FastServe.

Similar to previous experiments, we vary the rate, CV, and Zipfian parameter of the workload. In addition, we adjust the number of slots of the GPU key-value cache as an additional factor to evaluate the sensitivity of proactive key-value cache management to the cache size.

As shown in Figure 11(a), when the job arrival rate is low, the performance gap between FastServe and the two baselines is small, since the peak memory usage is low and the key-value cache is sufficient for all three solutions. As the job arrival rate grows, the peak memory usage exceeds



**Figure 12.** Scalability experiments under different number of NVIDIA A100 40GB GPUs when serving GPT-3 66B.

the GPU memory capacity, making FastServe significantly outperform Defer by up to 2.3×. For Reactive-offload, the high job arrival rate leads to more jobs that require key-value cache slots to arrive at the same time, but all need to wait for extra data transmission time. Therefore, FastServe is able to achieve 1.6× better performance than Reactive-offload. As for the impact of burstiness in Figure 11(b), FastServe also outperforms Defer and Reactive-offload by up to 3.5× and 1.4×, respectively, due to the overlapping between proactive swapping and computation. For the impact of skewness in Figure 11(c), the performance gap becomes larger as the Zipfian parameter  $\theta$  reduces. This is because the average job size becomes larger, due to the bounded maximum length and less skewness. Consequently, the peak memory usage increases, leading to more improvements of FastServe. As for Figure 11(d), when the number of key-value cache slots is small, the peak memory usage easily exceeds the key-value cache size, which requires more careful cache management. With overlapping the proactive swapping and computation, FastServe is able to outperform Defer and Reactive-offload by up to 1.8×. Note that when serving 175B-scale models, the cache size is greatly limited by the GPU memory capacity, making proactive swapping management a necessity.

#### 6.4 Scalability

In this subsection, we evaluate the scalability of FastServe for serving GPT-3 66B model. We vary the number of GPUs while fixing other parameters in the experiments to compare FastServe with FasterTransformer and Orca. The model is divided into two pipeline stages and tensor parallelism is adjusted accordingly based on the number of GPUs. Note that we do not use GPT-3 175B, because it needs at least 9 NVIDIA A100 40GB GPUs to just hold its weights and consumes more memory for the intermediate state during inference serving. GPT-3 66B can be served with only 6 NVIDIA A100 40GB GPUs, allowing us to vary the number of GPUs from 6 to 16 to evaluate the scalability. We report both average JCT and tail JCT (90% JCT) in the results. As shown in Figure 12, both average JCT and tail JCT decrease when more GPUs are used to serve inference jobs, as more computing resources speed up the execution time of each job with tensor parallelism.

With careful integration with distributed execution, Fast-Serve supports iter-job pipeline parallelism in its scheduler, and benefits from memory locality through its distributed key-value cache management. The results show that Fast-Serve achieves 3.5×–4× and 9.2×–11.1× improvement on average JCT than Orca and FasterTransformer, respectively. As for 90% tail JCT, FastServe outperforms them by 2.2×–6.4× and 4.3×–12.5×, respectively.

## 7 Related Work

**Preemptive scheduling.** Many solutions for job scheduling in datacenters use preemptive scheduling. PDQ [32], pFabric [6], Varys [16], and PIAS [8] use preemptive flow scheduling to minimize flow completion time. Shinjuku [36], Shenango[45], and Caladan [25] focus on latency-sensitive datacenter workloads, which use fine-grained preemption and resource reallocation to optimize for microsecond-scale tail latency. As for DL workloads, Tiresias [28] uses MLFQ to optimize JCT for distributed DL training jobs. Pipeswitch[10] and REEF[30] provide efficient GPU preemption to run both latency-critical and best-effort DL tasks to achieve both real-time and work conserving on GPU. By contrast, FastServe targets a new scenario, LLM inference serving, and is semi-information-agnostic.

**Inference serving.** TensorFlow Serving [43] and Triton Inference Server [19] are production-grade inference serving systems, which are widely used in industry. They serve as an abstraction above the execution engines and lack model-specific optimizations. Clipper [21], Clockwork [29], and Shepherd [59] focus on serving relatively small models like ResNet in a cluster and support latency-aware provision to maximize the overall goodput. INFaaS [46] proposes a model-less serving paradigm to automate the model selection, deployment, and serving process. There are also serving systems that incorporate domain-specific knowledge, such as Nexus [48] which targets DNN-based video analysis, and Inferline [20] which optimizes the serving pipeline that consists of multiple models. Recently, several serving systems are proposed to optimize Transformer-based LLMs [23, 38, 40, 58]. Orca [58] is the state-of-the-art solution that considers the autoregressive generation pattern of LLMs. However, its FCFS policy suffers from head-of-line blocking which we address in this paper.

**Memory management for LLMs.** Due to high memory usage for LLMs, many techniques have been proposed to reduce memory overhead. Some work [9, 54] targets training, which is orthogonal to the serving scenario. Quantization [22, 24, 39, 57] compresses the model weights into 8-bit or even 4-bit integers after training, which can greatly reduce the memory footprint during inference. Similarly, SparTA [60] is an end-to-end model sparsity framework to explore better sparse models. However, these approaches can

decrease the performance of the original model. Petals [11] runs the inference of LLMs in a collaborative fashion to amortize the cost via decentralization. Its performance is influenced due to network latency. Other works [7, 34, 49] use offloading to utilize host memory and disks. FlexGen [49] pushes this idea to support 175B-scale model with a single GPU. However, they all use a run-to-completion policy. To hide the data transmission time with computation, they target offline throughput-oriented applications which process a big batch at a time and are not suitable for interactive applications like ChatGPT. FastServe exploits preemption at the granularity of iteration to optimize for JCT.

## 8 Conclusion

We present FastServe, a distributed inference serving system for LLMs. We exploit the autoregressive pattern of LLM inference to enable iteration-level preemption and design a novel skip-join MLFQ scheduler to address head-of-line blocking. We propose a proactive key-value cache management mechanism to handle the memory overhead of the key-value cache and hide the data transmission latency with computing. Based on these techniques, we build a prototype of FastServe. Experiments show that FastServe improves the average JCT and tail JCT by up to 5.1 $\times$  and 6.4 $\times$  respectively, compared to the state-of-the-art solution Orca.

## References

- [1] 2022. Introducing ChatGPT. <https://openai.com/blog/chatgpt>. (2022).
- [2] 2023. Bard, an experiment by Google. <https://bard.google.com/>. (2023).
- [3] 2023. ChatGPT sets record for fastest-growing user base. <https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/>. (2023).
- [4] 2023. Hello Dolly: Democratizing the magic of ChatGPT with open models. <https://www.databricks.com/blog/2023/03/24/hello-dolly-democratizing-magic-chatgpt-open-models.html>. (2023).
- [5] 2023. Reinventing search with a new AI-powered Bing and Edge, your copilot for the web. <https://news.microsoft.com/the-new-Bing/>. (2023).
- [6] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. *SIGCOMM CCR* (2013).
- [7] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, et al. 2022. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. *arXiv* (2022).
- [8] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. Information-agnostic flow scheduling for commodity data centers. In *USENIX OSDI*.
- [9] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. 2021. Gradient Compression Supercharged High-Performance Data Parallel DNN Training. In *ACM SOSP*.
- [10] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. Pipeswitch: Fast pipelined context switching for deep learning applications. In *USENIX OSDI*.
- [11] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. 2022. Petals: Collaborative inference and fine-tuning of large models. *arXiv* (2022).
- [12] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. (2020).
- [13] Yanpei Chen, Sara Alspaugh, and Randy Katz. 2012. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *arXiv* (2012).
- [14] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%\* ChatGPT Quality. (2023). <https://vicuna.lmsys.org>
- [15] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient coflow scheduling without prior knowledge. *SIGCOMM CCR* (2015).
- [16] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with varys. In *ACM SIGCOMM*.
- [17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing*.
- [18] NVIDIA Corporation. 2019. FasterTransformer. (2019). <https://github.com/NVIDIA/FasterTransformer>
- [19] NVIDIA Corporation. 2019. Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution. (2019). <https://github.com/triton-inference-server/server>
- [20] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *ACM Symposium on Cloud Computing*.
- [21] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System.. In *USENIX NSDI*.
- [22] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv* (2022).
- [23] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: an efficient GPU serving system for transformer models. In *ACM PPoPP*.
- [24] Elias Frantar, Saleh Ashkboos, Torsten Hoefer, and Dan Alistarh. 2022. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. *arXiv* (2022).
- [25] Joshua Fried, Zhenyuan Ruan, Amy Oosterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *USENIX OSDI*.
- [26] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*.
- [27] Wanling Gao, Yuqing Zhu, Zhen Jia, Chunjie Luo, Lei Wang, Zhiguo Li, Jianfeng Zhan, Yong Qi, Yongqiang He, Shiming Gong, et al. 2013. Bigdatabench: a big data benchmark suite from web search engines. *arXiv* (2013).
- [28] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning.. In *USENIX NSDI*.
- [29] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like

- Clockwork: Performance Predictability from the Bottom Up. In *USENIX OSDI*.
- [30] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *USENIX OSDI*.
- [31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*.
- [32] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. 2012. Finishing Flows Quickly with Preemptive Scheduling. In *ACM SIGCOMM*.
- [33] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. (2019).
- [34] HuggingFace. 2022. Hugging face accelerate. (2022). <https://huggingface.co/docs/accelerate/index>
- [35] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for  $\mu$ second-scale Tail Latency. In *USENIX NSDI*.
- [36] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive scheduling for  $\mu$ second-scale tail latency. In *USENIX NSDI*.
- [37] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. (2020).
- [38] Dacheng Li, Rulin Shao, Hongyi Wang, Han Guo, Eric P. Xing, and Hao Zhang. 2023. MPCFormer: fast, performant and private Transformer inference with MPC. (2023).
- [39] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. 2020. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on Machine Learning (ICML)*.
- [40] Zhuohan Li, Lianmin Zheng, Yimin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. AlpAServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. *arXiv* (2023).
- [41] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *ACM SOSP*.
- [42] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. (2021).
- [43] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv* (2017).
- [44] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv* (2019).
- [45] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads.. In *USENIX NSDI*.
- [46] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *USENIX ATC*.
- [47] Linus Schrage. 1968. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research* (1968).
- [48] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *ACM SOSP*.
- [49] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. 2023. High-throughput Generative Inference of Large Language Models with a Single GPU. *arXiv* (2023).
- [50] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. (2020).
- [51] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. [https://github.com/tatsu-lab/stanford\\_alpaca](https://github.com/tatsu-lab/stanford_alpaca). (2023).
- [52] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. (2023).
- [53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Neural Information Processing Systems* (2017).
- [54] Jue Wang, Binhang Yuan, Luka Rimanic, Yongjun He, Tri Dao, Beidi Chen, Christopher Re, and Ce Zhang. 2023. Fine-tuning Language Models over Slow Networks using Activation Compression with Guarantees. (2023).
- [55] Alex Watson, Deepigha Shree Vittal Babu, and Suprio Ray. 2017. Sanzu: A data science benchmark. In *IEEE International Conference on Big Data*.
- [56] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierrick Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. HuggingFace’s Transformers: State-of-the-art Natural Language Processing. (2020).
- [57] Guangxuan Xiao, Ji Lin, Mickael Seznec, Julien Demouth, and Song Han. 2022. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv* (2022).
- [58] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for {Transformer-Based} Generative Models. In *USENIX OSDI*.
- [59] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. (2023).
- [60] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. 2022. SparTA: Deep-Learning Model Sparsity via Tensor-with-Sparsity-Attribute. In *USENIX OSDI*.



## Flash-Decoding for long-context inference

**Authors:** Tri Dao and Daniel Haziza and Francisco Massa and Grigory Sizov

### Motivation

Large language models (LLM) such as ChatGPT or Llama have received unprecedented attention lately. However, they remain massively expensive to run. Even though generating a single response can cost about \$0.01 (a few seconds of an 8xA100 instance on AWS), the costs quickly add up when scaling to billions of users, who could have multiple daily interactions with such LLMs. Some use cases are more expensive, like code auto-completion, because it runs whenever a new character is typed. As LLM applications multiply, even small efficiency gains to the generation time can have a massive impact.

LLM inference (or “decoding”) is an iterative process: tokens are generated one at a time. Generating full sentences of  $N$  tokens requires  $N$  forward passes through the model. Fortunately, it is possible to cache previously calculated tokens: this means that a single generation step does not depend on the context length, except for a single operation, the attention. This operation does not scale well with context length.

There are a number of important emerging use cases of LLMs that utilize a long context. With a longer context, LLMs can reason about longer documents, either to summarize or answer questions about them, they can keep track of longer conversations, or even process entire codebases before writing code. As an example, most LLMs had a context length of up to 2k in 2022 (GPT-3), but we now have open-source LLMs scaling up to 32k (Llama-2-32k), or even 100k more recently (CodeLlama). In this setting, attention takes a significant fraction of time during inference.

When scaling on the batch size dimension, the attention can also become a bottleneck even with relatively small contexts. This is because the amount of memory to read scales with the batch dimension, whereas it only depends on the model size for the rest of the model.

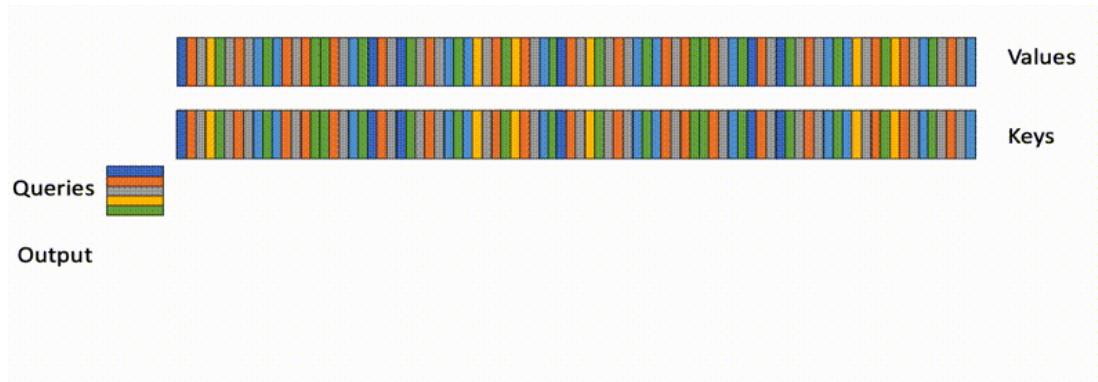
We present a technique, Flash-Decoding, that significantly speeds up attention during inference, bringing up to 8x faster generation for very long sequences. The main idea is to load the research values in parallel as possible, then separately rescale and combine results to maintain the right attention outputs.



## Multi-head attention for decoding

During decoding, every new token that is generated needs to attend to all previous tokens, to compute:  $\text{softmax}(\text{queries} @ \text{keys.transpose}) @ \text{values}$

This operation has been optimized with FlashAttention (v1 and v2 recently) in the training case, where the bottleneck is the memory bandwidth to read and write the intermediate results (e.g.  $\text{Q} @ \text{K}^T$ ). However, these optimizations don't apply directly to the inference case, because the bottlenecks are different. For training, FlashAttention parallelizes across the batch size and query length dimensions. During inference, the query length is typically 1: this means that if the batch size is smaller than the number of streaming multiprocessors (SMs) on the GPU (108 for an A100), the operation will only use a small part of the GPU! This is especially the case when using long contexts, because it requires smaller batch sizes to fit in GPU memory. With a batch size of 1, FlashAttention will use less than 1% of the GPU!

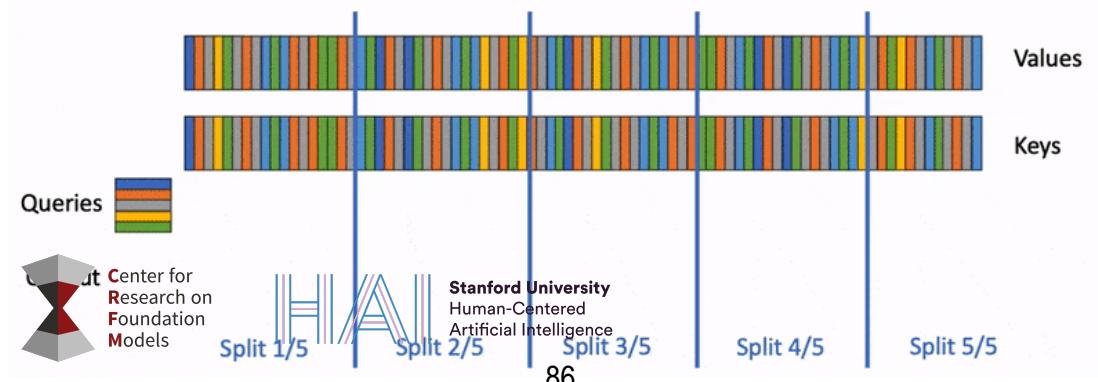


FlashAttention parallelizes across blocks of queries and batch size only, and does not manage to occupy the entire GPU during decoding

The attention can also be done using matrix multiplication primitives - without using FlashAttention. In this case, the operation occupies the GPU entirely, but launches many kernels that write and read intermediate results, which is not optimal.

## A faster attention for decoding: Flash-Decoding

Our new approach Flash-Decoding is based on FlashAttention, and adds a new parallelization dimension: the keys/values sequence length. It combines the benefits of the 2 approaches from above. Like FlashAttention, it stores very little extra data to global memory, however it fully utilizes the GPU even when the batch size is small, as long as the context length is large enough.



Flash-Decoding also parallelizes across keys and values, at the cost of a small final reduction step

Flash-Decoding works in 3 steps:

1. First, we split the keys/values in smaller chunks
2. We compute the attention of the query with each of these splits in parallel using FlashAttention. We also write 1 extra scalar per row and per split: the log-sum-exp of the attention values.
3. Finally, we compute the actual output by reducing over all the splits, using the log-sum-exp to scale the contribution of each split.

All of this is possible because the attention/softmax can be calculated iteratively. In Flash-Decoding, it is used at 2 levels: within splits (like FlashAttention), and across splits to perform the final reduction.

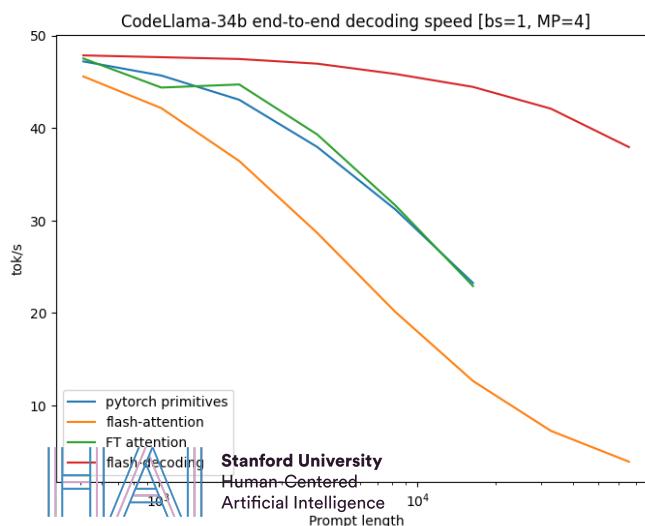
In practice, step (1) does not involve any GPU operation, as the key/value chunks are views of the full key/value tensors. We then have 2 separate kernels to perform respectively (2) and (3).

## Benchmarks on CodeLlama 34B

To validate this approach, we benchmark the decoding throughput of the CodeLLaMa-34b. This model has the same architecture as Llama 2, and more generally results should generalize across many LLMs. We measure the decoding speed in tok/s at various sequence lengths, from 512 to 64k, and compare multiple ways of calculating the attention:

- Pytorch: Running the attention using pure PyTorch primitives (without using FlashAttention)
- FlashAttention v2 (before v2.2)
- FasterTransformer: Uses the FasterTransformer attention kernel
- Flash-Decoding
- And an upper bound calculated as the time it takes to read from memory the entire model along with the KV-cache

Flash-Decoding unlocks up to 8x speedups in decoding speed for very large sequences, and scales much better than alternative approaches.



All approaches perform similarly for small prompts, but scale poorly as the sequence length increases from 512 to 64k, except Flash-Decoding. In this regime (batch size 1) with Flash-Decoding, scaling the sequence length has little impact on generation speed.

## Component-level micro-benchmarks

We also micro-benchmark the scaled multi-head attention for various sequence lengths and batch sizes on A100 with inputs in f16. We set the batch size to 1, and use 16 query heads of dimension 128, for 2 key/value heads (grouped-query attention), which matches the dimensions used in CodeLLaMa-34b when running on 4 GPUs.

Setting \ Algorithm	PyTorch Eager	Flash-Attention v2.0.9	Flash-Decoding
B=256, seqlen=256	3058.6	390.5	63.4
B=128, seqlen=512	3151.4	366.3	67.7
B=64, seqlen=1024	3160.4	364.8	77.7
B=32, seqlen=2048	3158.3	352	58.5
B=16, seqlen=4096	3157	401.7	57
B=8, seqlen=8192	3173.1	529.2	56.4
B=4, seqlen=16384	3223	582.7	58.2
B=2, seqlen=32768	3224.1	1156.1	60.3
B=1, seqlen=65536	1335.6	2300.6	64.4
B=1, seqlen=131072	2664	4592.2	106.6

Micro-benchmark of the multi-head attention, run-time in us. Flash-Decoding achieves almost constant run-time as the sequence length scales to up to 64k.

The up to 8x speedup end-to-end measured earlier is made possible because the attention itself is up to 50x faster than FlashAttention. Up until sequence length 32k, the attention time is roughly constant, because Flash-Decoding manages to fully utilize the GPU.

## Using Flash-Decoding

Flash-decoding is available:

- In the FlashAttention package, starting at version 2.2.
- Through xFormers starting at version 0.0.22 through `xformers.ops.memory_efficient_attention`. The dispatcher will automatically use either the Flash-Decoding or FlashAttention approaches depending on the problem size. When these approaches are not supported, it can dispatch to an efficient triton kernel that implements the Flash-Decoding algorithm.

A full example of decoding with LLaMa v2 / CodeLLaMa is available in the FlashAttention repo [here](#) and in the xFormers repo [here](#). We also provide a [minimal example](#) of an efficient decoding code for LLaMa v1/v2 models, meant to be fast, easy to read, educational and hackable.

## Acknowledgment

Thanks to Erich Elsen, Ashish Vaswani, and Michaël Benesty for suggesting this idea of splitting the KVcache loading. We want to thank Jeremy Reizenstein, Patrick Labatut and Andrew Tulloch for the valuable discussions. We also want to thank Geeta Chauhan and Gregory Chanan for helping with the writing.



Sign up to get email updates on the Center for Research on Foundation Models (CRFM) or  
email us at [contact-crfm@stanford.edu](mailto:contact-crfm@stanford.edu).

CRFM is grateful to our supporters.

© 2021. Stanford Center for Research on Foundation Models.

Designed by Joon Sung Park.



# FLASHATTENTION: Fast and Memory-Efficient Exact Attention with IO-Awareness

Tri Dao<sup>†</sup>, Daniel Y. Fu<sup>†</sup>, Stefano Ermon<sup>†</sup>, Atri Rudra<sup>‡</sup>, and Christopher Ré<sup>†</sup>

<sup>†</sup>Department of Computer Science, Stanford University

<sup>‡</sup>Department of Computer Science and Engineering, University at Buffalo, SUNY

{trid, danfu}@cs.stanford.edu, ermon@stanford.edu, atri@buffalo.edu,  
chrismre@cs.stanford.edu

June 24, 2022

## Abstract

Transformers are slow and memory-hungry on long sequences, since the time and memory complexity of self-attention are quadratic in sequence length. Approximate attention methods have attempted to address this problem by trading off model quality to reduce the compute complexity, but often do not achieve wall-clock speedup. We argue that a missing principle is making attention algorithms *IO-aware*—accounting for reads and writes between levels of GPU memory. We propose FLASHATTENTION, an IO-aware exact attention algorithm that uses tiling to reduce the number of memory reads/writes between GPU high bandwidth memory (HBM) and GPU on-chip SRAM. We analyze the IO complexity of FLASHATTENTION, showing that it requires fewer HBM accesses than standard attention, and is optimal for a range of SRAM sizes. We also extend FLASHATTENTION to block-sparse attention, yielding an approximate attention algorithm that is faster than any existing approximate attention method. FLASHATTENTION trains Transformers faster than existing baselines: 15% end-to-end wall-clock speedup on BERT-large (seq. length 512) compared to the MLPerf 1.1 training speed record, 3× speedup on GPT-2 (seq. length 1K), and 2.4× speedup on long-range arena (seq. length 1K-4K). FLASHATTENTION and block-sparse FLASHATTENTION enable longer context in Transformers, yielding higher quality models (0.7 better perplexity on GPT-2 and 6.4 points of lift on long-document classification) and entirely new capabilities: the first Transformers to achieve better-than-chance performance on the Path-X challenge (seq. length 16K, 61.4% accuracy) and Path-256 (seq. length 64K, 63.1% accuracy).

## 1 Introduction

Transformer models [82] have emerged as the most widely used architecture in applications such as natural language processing and image classification. Transformers have grown larger [5] and deeper [83], but equipping them with longer context remains difficult [80], since the self-attention module at their heart has time and memory complexity quadratic in sequence length. An important question is whether making attention faster and more memory-efficient can help Transformer models address their runtime and memory challenges for long sequences.

Many approximate attention methods have aimed to reduce the compute and memory requirements of attention. These methods range from sparse-approximation [51, 74] to low-rank approximation [12, 50, 84], and their combinations [3, 9, 92]. Although these methods reduce the compute requirements to linear or near-linear in sequence length, many of them do not display wall-clock speedup against standard attention and have not gained wide adoption. One main reason is that they focus on FLOP reduction (which may not correlate with wall-clock speed) and tend to ignore overheads from memory access (IO).

In this paper, we argue that a missing principle is making attention algorithms *IO-aware* [1]—that is, carefully accounting for reads and writes to different levels of fast and slow memory (e.g., between fast GPU on-chip SRAM and relatively slow GPU high bandwidth memory, or HBM [45], Figure 1 left). On modern

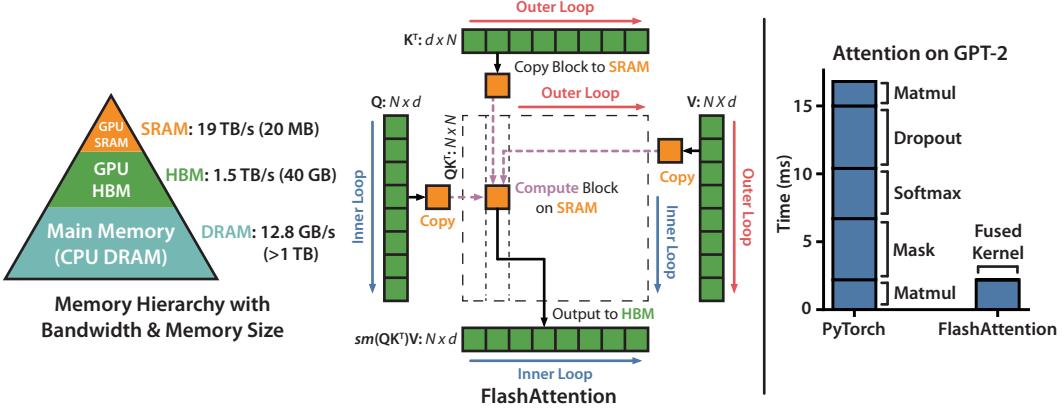


Figure 1: **Left:** FLASHATTENTION uses tiling to prevent materialization of the large  $N \times N$  attention matrix (dotted box) on (relatively) slow GPU HBM. In the outer loop (red arrows), FLASHATTENTION loops through blocks of the  $\mathbf{K}$  and  $\mathbf{V}$  matrices and loads them to fast on-chip SRAM. In each block, FLASHATTENTION loops over blocks of  $\mathbf{Q}$  matrix (blue arrows), loading them to SRAM, and writing the output of the attention computation back to HBM. **Right:** Speedup over the PyTorch implementation of attention on GPT-2. FLASHATTENTION does not read and write the large  $N \times N$  attention matrix to HBM, resulting in an 7.6× speedup on the attention computation.

GPUs, compute speed has out-paced memory speed [61, 62, 63], and most operations in Transformers are bottlenecked by memory accesses [43]. IO-aware algorithms have been critical for similar memory-bound operations, when reading and writing data can account for a large portion of the runtime—such as database joins [71], image processing [70], numerical linear algebra [4], and more [40, 85]. However, common Python interfaces to deep learning such as PyTorch and Tensorflow do not allow fine-grained control of memory access.

We propose FLASHATTENTION, a new attention algorithm that computes exact attention with far fewer memory accesses. Our main goal is to avoid reading and writing the attention matrix to and from HBM. This requires (i) computing the softmax reduction without access to the whole input (ii) not storing the large intermediate attention matrix for the backward pass. We apply two well-established techniques to address these challenges. (i) We restructure the attention computation to split the input into blocks and make several passes over input blocks, thus incrementally performing the softmax reduction (also known as **tiling**). (ii) We store the softmax normalization factor from the forward pass to quickly **recompute** attention on-chip in the backward pass, which is faster than the standard approach of reading the intermediate attention matrix from HBM. We implement FLASHATTENTION in CUDA to achieve fine-grained control over memory access and fuse all the attention operations into one GPU kernel. Even with the increased FLOPs due to recomputation, our algorithm both **runs faster** (up to 7.6x on GPT-2 [67], Figure 1 right) and **uses less memory**—linear in sequence length—than standard attention, thanks to the massively reduced amount of HBM access.

We analyze the IO complexity [1] of FLASHATTENTION, proving that it requires  $O(N^2 d^2 M^{-1})$  HBM accesses where  $d$  is the head dimension and  $M$  is the size of SRAM, as compared to  $\Omega(Nd + N^2)$  of standard attention. For typical values of  $d$  and  $M$ , FLASHATTENTION requires many times fewer HBM accesses compared to standard attention (up to 9x fewer, as shown in Fig. 2). Moreover, we provide a lower bound, showing that no exact attention algorithm can asymptotically improve on the number of HBM accesses over all SRAM sizes.

We also show that FLASHATTENTION can serve as a useful primitive for realizing the potential of approximate attention algorithms by overcoming their issues with memory access overhead. As a proof of concept, we implement block-sparse FLASHATTENTION, a sparse attention algorithm that is 2-4x faster than even FLASHATTENTION, scaling up to sequence length of 64k. We prove that block-sparse FLASHATTENTION has better IO complexity than FLASHATTENTION by a factor proportional to the sparsity ratio. We discuss further extensions to other operations (attention on multi-GPU, kernel regression, block-sparse matrix

multiply) in Section 5. We open-source FLASHATTENTION to make it easier to build on this primitive.<sup>1</sup>

We empirically validate that FLASHATTENTION speeds up model training and improves model quality by modeling longer context. We also benchmark the runtime and memory footprint of FLASHATTENTION and block-sparse FLASHATTENTION compared to prior attention implementations.

- **Faster Model Training.** FLASHATTENTION trains Transformer models faster in wall-clock time. We train BERT-large (seq. length 512) 15% faster than the training speed record in MLPerf 1.1 [58], GPT2 (seq. length 1K) 3× faster than baseline implementations from HuggingFace [87] and Megatron-LM [77], and long-range arena (seq. length 1K-4K) 2.4× faster than baselines.
- **Higher Quality Models.** FLASHATTENTION scales Transformers to longer sequences, which improves their quality and enables new capabilities. We observe a 0.7 improvement in perplexity on GPT-2 and 6.4 points of lift from modeling longer sequences on long-document classification [13]. FLASHATTENTION enables the first Transformer that can achieve better-than-chance performance on the Path-X [80] challenge, solely from using a longer sequence length (16K). Block-sparse FLASHATTENTION enables a Transformer to scale to even longer sequences (64K), resulting in the first model that can achieve better-than-chance performance on Path-256.
- **Benchmarking Attention.** FLASHATTENTION is up to 3× faster than the standard attention implementation across common sequence lengths from 128 to 2K and scales up to 64K. Up to sequence length of 512, FLASHATTENTION is both faster and more memory-efficient than any existing attention method, whereas for sequence length beyond 1K, some approximate attention methods (e.g., Linformer) start to become faster. On the other hand, block-sparse FLASHATTENTION is faster than all existing approximate attention methods that we know of.

## 2 Background

We provide some background on the performance characteristics of common deep learning operations on modern hardware (GPUs). We also describe the standard implementation of attention.

### 2.1 Hardware Performance

We focus here on GPUs. Performance on other hardware accelerators are similar [46, 48].

**GPU Memory Hierarchy.** The GPU memory hierarchy (Fig. 1 left) comprises multiple forms of memory of different sizes and speeds, with smaller memory being faster. As an example, the A100 GPU has 40-80GB of high bandwidth memory (HBM) with bandwidth 1.5-2.0TB/s and 192KB of on-chip SRAM per each of 108 streaming multiprocessors with bandwidth estimated around 19TB/s [44, 45]. The on-chip SRAM is an order of magnitude faster than HBM but many orders of magnitude smaller in size. As compute has gotten faster relative to memory speed [61, 62, 63], operations are increasingly bottlenecked by memory (HBM) accesses. Thus exploiting fast SRAM becomes more important.

**Execution Model.** GPUs have a massive number of threads to execute an operation (called a kernel). Each kernel loads inputs from HBM to registers and SRAM, computes, then writes outputs to HBM.

**Performance characteristics.** Depending on the balance of computation and memory accesses, operations can be classified as either compute-bound or memory-bound. This is commonly measured by the *arithmetic intensity* [85], which is the number of arithmetic operations per byte of memory access.

1. Compute-bound: the time taken by the operation is determined by how many arithmetic operations there are, while time accessing HBM is much smaller. Typical examples are matrix multiply with large inner dimension, and convolution with large number of channels.
2. Memory-bound: the time taken by the operation is determined by the number of memory accesses, while time spent in computation is much smaller. Examples include most other operations: elementwise (e.g., activation, dropout), and reduction (e.g., sum, softmax, batch norm, layer norm).

**Kernel fusion.** The most common approach to accelerate memory-bound operations is kernel fusion: if there are multiple operations applied to the same input, the input can be loaded once from HBM, instead of multiple times for each operation. Compilers can automatically fuse many elementwise operations [53, 65, 75].

---

<sup>1</sup>FLASHATTENTION code is available at <https://github.com/HazyResearch/flash-attention>

However, in the context of model training, the intermediate values still need to be written to HBM to save for the backward pass, reducing the effectiveness of naive kernel fusion.

## 2.2 Standard Attention Implementation

Given input sequences  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$  where  $N$  is the sequence length and  $d$  is the head dimension, we want to compute the attention output  $\mathbf{O} \in \mathbb{R}^{N \times d}$ :

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \text{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d},$$

where softmax is applied row-wise.

Standard attention implementations materialize the matrices  $\mathbf{S}$  and  $\mathbf{P}$  to HBM, which takes  $O(N^2)$  memory. Often  $N \gg d$  (e.g., for GPT2,  $N = 1024$  and  $d = 64$ ). We describe the standard attention implementation in Algorithm 0. As some or most of the operations are memory-bound (e.g., softmax), the large number of memory accesses translates to slow wall-clock time.

This problem is exacerbated by other elementwise operations applied to the attention matrix, such as masking applied to  $\mathbf{S}$  or dropout applied to  $\mathbf{P}$ . As a result, there have been many attempts to fuse several elementwise operations, such as fusing masking with softmax [77].

In Section 3.2, we will show that the standard attention implementation performs HBM accesses quadratic in the sequence length  $N$ . We also compare the number of FLOPs and number of HBM accesses of standard attention and of our method (FLASHATTENTION).

---

### Algorithm 0 Standard Attention Implementation

---

**Require:** Matrices  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$  in HBM.

- 1: Load  $\mathbf{Q}, \mathbf{K}$  by blocks from HBM, compute  $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$ , write  $\mathbf{S}$  to HBM.
  - 2: Read  $\mathbf{S}$  from HBM, compute  $\mathbf{P} = \text{softmax}(\mathbf{S})$ , write  $\mathbf{P}$  to HBM.
  - 3: Load  $\mathbf{P}$  and  $\mathbf{V}$  by blocks from HBM, compute  $\mathbf{O} = \mathbf{P}\mathbf{V}$ , write  $\mathbf{O}$  to HBM.
  - 4: Return  $\mathbf{O}$ .
- 

## 3 FLASHATTENTION: Algorithm, Analysis, and Extensions

We show how to compute exact attention with fewer HBM reads/writes and without storing large intermediate matrices for the backward pass. This yields an attention algorithm that is both memory efficient and faster in wall-clock time. We analyze its IO complexity, showing that our method requires much fewer HBM accesses compared to standard attention. We further show that FLASHATTENTION can serve as a useful primitive by extending it to handle block-sparse attention.

We focus here on the forward pass for ease of exposition; Appendix B contains details for the backward.

### 3.1 An Efficient Attention Algorithm With Tiling and Recomputation

Given the inputs  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$  in HBM, we aim to compute the attention output  $\mathbf{O} \in \mathbb{R}^{N \times d}$  and write it to HBM. Our goal is to reduce the amount of HBM accesses (to sub-quadratic in  $N$ ).

We apply two established techniques (tiling, recomputation) to overcome the technical challenge of computing exact attention in sub-quadratic HBM accesses. We describe this in Algorithm 1. The main idea is that we split the inputs  $\mathbf{Q}, \mathbf{K}, \mathbf{V}$  into blocks, load them from slow HBM to fast SRAM, then compute the attention output with respect to those blocks. By scaling the output of each block by the right normalization factor before adding them up, we get the correct result at the end.

**Tiling.** We compute attention by blocks. Softmax couples columns of  $\mathbf{K}$ , so we decompose the large softmax with scaling [51, 60, 66]. For numerical stability, the softmax of vector  $x \in \mathbb{R}^B$  is computed as:

$$m(x) := \max_i x_i, \quad f(x) := [e^{x_1 - m(x)} \quad \dots \quad e^{x_B - m(x)}], \quad \ell(x) := \sum_i f(x)_i, \quad \text{softmax}(x) := \frac{f(x)}{\ell(x)}.$$

For vectors  $x^{(1)}, x^{(2)} \in \mathbb{R}^B$ , we can decompose the softmax of the concatenated  $x = [x^{(1)} \ x^{(2)}] \in \mathbb{R}^{2B}$  as:

$$m(x) = m([x^{(1)} \ x^{(2)}]) = \max(m(x^{(1)}), m(x^{(2)})), \quad f(x) = \begin{bmatrix} e^{m(x^{(1)})-m(x)} f(x^{(1)}) & e^{m(x^{(2)})-m(x)} f(x^{(2)}) \end{bmatrix},$$

$$\ell(x) = \ell([x^{(1)} \ x^{(2)}]) = e^{m(x^{(1)})-m(x)} \ell(x^{(1)}) + e^{m(x^{(2)})-m(x)} \ell(x^{(2)}), \quad \text{softmax}(x) = \frac{f(x)}{\ell(x)}.$$

Therefore if we keep track of some extra statistics  $(m(x), \ell(x))$ , we can compute softmax one block at a time.<sup>2</sup> We thus split the inputs  $\mathbf{Q}, \mathbf{K}, \mathbf{V}$  into blocks (Algorithm 1 line 3), compute the softmax values along with extra statistics (Algorithm 1 line 10), and combine the results (Algorithm 1 line 12).

**Recomputation.** One of our goals is to not store  $O(N^2)$  intermediate values for the backward pass. The backward pass typically requires the matrices  $\mathbf{S}, \mathbf{P} \in \mathbb{R}^{N \times N}$  to compute the gradients with respect to  $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ . However, by storing the output  $\mathbf{O}$  and the softmax normalization statistics  $(m, \ell)$ , we can recompute the attention matrix  $\mathbf{S}$  and  $\mathbf{P}$  easily in the backward pass from blocks of  $\mathbf{Q}, \mathbf{K}, \mathbf{V}$  in SRAM. This can be seen as a form of selective gradient checkpointing [10, 34]. While gradient checkpointing has been suggested to reduce the maximum amount of memory required [66], all implementations (that we know off) have to trade speed for memory. In contrast, even with more FLOPs, our recomputation speeds up the backward pass due to reduced HBM accesses (Fig. 2). The full backward pass description is in Appendix B.

**Implementation details: Kernel fusion.** Tiling enables us to implement our algorithm in one CUDA kernel, loading input from HBM, performing all the computation steps (matrix multiply, softmax, optionally masking and dropout, matrix multiply), then write the result back to HBM (masking and dropout in Appendix B). This avoids repeatedly reading and writing of inputs and outputs from and to HBM.

---

#### Algorithm 1 FLASHATTENTION

---

**Require:** Matrices  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$  in HBM, on-chip SRAM of size  $M$ .

- 1: Set block sizes  $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$ .
  - 2: Initialize  $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$  in HBM.
  - 3: Divide  $\mathbf{Q}$  into  $T_r = \lceil \frac{N}{B_r} \rceil$  blocks  $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$  of size  $B_r \times d$  each, and divide  $\mathbf{K}, \mathbf{V}$  in to  $T_c = \lceil \frac{N}{B_c} \rceil$  blocks  $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$  and  $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$ , of size  $B_c \times d$  each.
  - 4: Divide  $\mathbf{O}$  into  $T_r$  blocks  $\mathbf{O}_1, \dots, \mathbf{O}_{T_r}$  of size  $B_r \times d$  each, divide  $\ell$  into  $T_r$  blocks  $\ell_1, \dots, \ell_{T_r}$  of size  $B_r$  each, divide  $m$  into  $T_r$  blocks  $m_1, \dots, m_{T_r}$  of size  $B_r$  each.
  - 5: **for**  $1 \leq j \leq T_c$  **do**
  - 6:   Load  $\mathbf{K}_j, \mathbf{V}_j$  from HBM to on-chip SRAM.
  - 7:   **for**  $1 \leq i \leq T_r$  **do**
  - 8:     Load  $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$  from HBM to on-chip SRAM.
  - 9:     On chip, compute  $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$ .
  - 10:    On chip, compute  $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}, \tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$  (pointwise),  $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$ .
  - 11:    On chip, compute  $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}, \ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$ .
  - 12:    Write  $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$  to HBM.
  - 13:    Write  $\ell_i \leftarrow \ell_i^{\text{new}}, m_i \leftarrow m_i^{\text{new}}$  to HBM.
  - 14:   **end for**
  - 15: **end for**
  - 16: Return  $\mathbf{O}$ .
- 

We show FLASHATTENTION’s correctness, runtime, and memory requirement (proof in Appendix C).

**Theorem 1.** Algorithm 1 returns  $\mathbf{O} = \text{softmax}(\mathbf{Q}\mathbf{K}^\top)\mathbf{V}$  with  $O(N^2d)$  FLOPs and requires  $O(N)$  additional memory beyond inputs and output.

## 3.2 Analysis: IO Complexity of FLASHATTENTION

We analyze the IO complexity of FLASHATTENTION, showing significant reduction in HBM accesses compared to standard attention. We also provide a lower bound, proving that no exact attention algorithm can

---

<sup>2</sup>This style of aggregation is called *algebraic aggregation* [33].

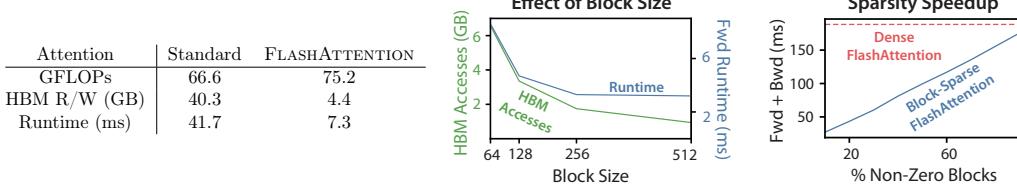


Figure 2: **Left:** Forward + backward runtime of standard attention and FLASHATTENTION for GPT-2 medium (seq. length 1024, head dim. 64, 16 heads, batch size 64) on A100 GPU. HBM access is the primary factor affecting runtime. **Middle:** Forward runtime of FLASHATTENTION (seq. length 1024, head dim. 64, 16 heads, batch size 64) on A100 GPU. Fewer HBM accesses result in faster runtime, up to a point. **Right:** The runtime (for seq. length 4K) of block-sparse FLASHATTENTION is faster than FLASHATTENTION by a factor proportional to the sparsity.

asymptotically improve on HBM accesses over all SRAM sizes. Proofs are in Appendix C.

**Theorem 2.** Let  $N$  be the sequence length,  $d$  be the head dimension, and  $M$  be size of SRAM with  $d \leq M \leq Nd$ . Standard attention (Algorithm 0) requires  $\Theta(Nd + N^2)$  HBM accesses, while FLASHATTENTION (Algorithm 1) requires  $\Theta(N^2d^2M^{-1})$  HBM accesses.

For typical values of  $d$  (64-128) and  $M$  (around 100KB),  $d^2$  is many times smaller than  $M$ , and thus FLASHATTENTION requires many times fewer HBM accesses than standard implementation. This leads to both faster execution and lower memory footprint, which we validate in Section 4.3.

The main idea of the proof is that given the SRAM size of  $M$ , we can load blocks of  $\mathbf{K}, \mathbf{V}$  of size  $\Theta(M)$  each (Algorithm 1 line 6). For each block of  $\mathbf{K}$  and  $\mathbf{V}$ , we iterate over all blocks of  $\mathbf{Q}$  (Algorithm 1 line 8) to compute the intermediate values, resulting in  $\Theta(NdM^{-1})$  passes over  $\mathbf{Q}$ . Each pass loads  $\Theta(Nd)$  elements, which amounts to  $\Theta(N^2d^2M^{-1})$  HBM accesses. We similarly prove that the backward pass of standard attention requires  $\Theta(Nd + N^2)$  HBM accesses while the backward pass of FLASHATTENTION requires  $\Theta(N^2d^2M^{-1})$  HBM accesses (Appendix B).

We prove a lower-bound: one cannot asymptotically improve on the number of HBM accesses for all values of  $M$  (the SRAM size) when computing exact attention.

**Proposition 3.** Let  $N$  be the sequence length,  $d$  be the head dimension, and  $M$  be size of SRAM with  $d \leq M \leq Nd$ . There does not exist an algorithm to compute exact attention with  $o(N^2d^2M^{-1})$  HBM accesses for all  $M$  in the range  $[d, Nd]$ .

The proof relies on the fact that for  $M = \Theta(Nd)$  any algorithm must perform  $\Omega(N^2d^2M^{-1}) = \Omega(Nd)$  HBM accesses. This type of lower bound over a subrange of  $M$  is common in the streaming algorithms literature [88]. We leave proving parameterized complexity [27] lower bounds in terms of  $M$  as exciting future work.

We validate that the number of HBM accesses is the main determining factor of attention run-time. In Fig. 2 (left), we see that even though FLASHATTENTION has higher FLOP count compared to standard attention (due to recomputation in the backward pass), it has much fewer HBM accesses, resulting in much faster runtime. In Fig. 2 (middle), we vary the block size  $B_c$  of FLASHATTENTION, which results in different amounts of HBM accesses, and measure the runtime of the forward pass. As block size increases, the number of HBM accesses decreases (as we make fewer passes over the input), and runtime decreases. For large enough block size (beyond 256), the runtime is then bottlenecked by other factors (e.g., arithmetic operations). Moreover, larger block size will not fit into the small SRAM size.

### 3.3 Extension: Block-Sparse FLASHATTENTION

We extend FLASHATTENTION to approximate attention: we propose block-sparse FLASHATTENTION, whose IO complexity is smaller than FLASHATTENTION by a factor proportional to the sparsity.

Given inputs  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$  and a mask matrix  $\tilde{\mathbf{M}} \in \{0, 1\}^{N \times N}$ , we want to compute:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \text{softmax}(\mathbf{S} \odot \mathbf{1}_{\tilde{\mathbf{M}}}) \in \mathbb{R}^{N \times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d},$$

where  $(\mathbf{S} \odot \mathbf{1}_{\tilde{\mathbf{M}}})_{kl} = \mathbf{S}_{kl}$  if  $\tilde{\mathbf{M}}_{kl} = 1$  and  $-\infty$  if  $\tilde{\mathbf{M}}_{kl} = 0$ . We require  $\tilde{\mathbf{M}}$  to have block form: for some block sizes  $B_r, B_c$ , for all  $k, l$ ,  $\tilde{\mathbf{M}}_{k,l} = \mathbf{M}_{ij}$  with  $i = \lfloor k/B_r \rfloor$ ,  $j = \lfloor l/B_c \rfloor$  for some  $\mathbf{M} \in \{0, 1\}^{N/B_r \times N/B_c}$ .

Given a predefined block sparsity mask  $\mathbf{M} \in \{0, 1\}^{N/B_r \times N/B_c}$  we can easily adapt Algorithm 1 to only compute the nonzero blocks of the attention matrix. The algorithm is identical to Algorithm 1, except we skip zero blocks. We reproduce the algorithm description in Algorithm 5 in Appendix B.

We also analyze the IO complexity of block-sparse FLASHATTENTION.

**Proposition 4.** *Let  $N$  be the sequence length,  $d$  be the head dimension, and  $M$  be size of SRAM with  $d \leq M \leq Nd$ . Block-sparse FLASHATTENTION (Algorithm 5) requires  $\Theta(Nd + N^2 d^2 M^{-1}s)$  HBM accesses where  $s$  is the fraction of nonzero blocks in the block-sparsity mask.*

We see that applying block-sparsity yields a direct improvement by the sparsity to the larger term in the IO complexity. For large sequence lengths  $N$ ,  $s$  is often set to  $N^{-1/2}$  [11] or  $N^{-1} \log N$  [3, 17, 92], resulting in  $\Theta(N\sqrt{N})$  or  $\Theta(N \log N)$  IO complexity. For downstream experiments, we use the fixed butterfly sparsity pattern [17], which has been shown to be able to approximate arbitrary sparsity [16].

In Fig. 2 (right), we validate that as the sparsity increases, the runtime of block-sparse FLASHATTENTION improves proportionally. On the LRA benchmark, block-sparse FLASHATTENTION achieves 2.8 $\times$  speedup, while performing on par with standard attention (Section 4).

## 4 Experiments

We evaluate the impact of using FLASHATTENTION to train Transformer models. We validate two claims about training time and model accuracy, and report attention runtime and memory benchmarks.

- **Training Speed.** FLASHATTENTION outperforms the MLPerf 1.1 [58] speed record for BERT by 15%, and speeds up GPT-2 up to 3 $\times$  over HuggingFace [87] and 1.8 $\times$  over Megatron [77] over standard Transformers. FLASHATTENTION speeds up the long-range arena (LRA) benchmark 2.4 $\times$ .
- **Quality.** FLASHATTENTION scales Transformers to longer sequences, yielding higher quality. FLASHATTENTION trains GPT-2 with context length 4K faster than Megatron trains GPT-2 with context length 1K, while achieving 0.7 better perplexity. Modeling longer sequences yields 6.4 points of lift on two long-document classification tasks. Finally, FLASHATTENTION yields the **first Transformer** that can achieve better-than-random performance on the challenging Path-X task (sequence length 16K), and block-sparse FLASHATTENTION yields the **first sequence model** that we know of that can achieve better-than-random performance on Path-256 (sequence length 64K).
- **Benchmarking Attention.** We measure the runtime and memory performance of FLASHATTENTION and block-sparse FLASHATTENTION based on sequence length. We confirm that the memory footprint of FLASHATTENTION scales linearly with seq. length and is up to 3 $\times$  faster than standard attention for common seq. lengths (up to 2K). We confirm that runtime of block-sparse FLASHATTENTION scales linearly in seq. length and is faster than all existing approximate attention baselines.

Additional experiment details are in Appendix E.

### 4.1 Faster Models with FLASHATTENTION

**BERT.** FLASHATTENTION yields the fastest single-node BERT training speed that we know of. We train a BERT-large [22] model with FLASHATTENTION on Wikipedia. Table 1 compares our training time to the implementation from Nvidia that set the training speed record for MLPerf 1.1 [58]. Our implementation is 15% faster.

Table 1: Training time of BERT-large, starting from the same initialization provided by the MLPerf benchmark, to reach the target accuracy of 72.0% on masked language modeling. Averaged over 10 runs on 8 $\times$ A100 GPUs.

BERT Implementation	Training time (minutes)
Nvidia MLPerf 1.1 [58]	$20.0 \pm 1.5$
FLASHATTENTION (ours)	<b><math>17.4 \pm 1.4</math></b>

**GPT-2.** FLASHATTENTION yields faster training times for GPT-2 [67] on the large OpenWebtext dataset [32] than the widely used HuggingFace [87] and Megatron-LM [77] implementations. Table 2 shows up to 3 $\times$  end-to-end speedup compared to Huggingface and 1.7 $\times$  speedup compared to Megatron-LM. FLASHATTENTION

achieves the same perplexity as the other two implementations, as we do not change the model definition. Appendix E includes plots of the validation perplexity throughout training, confirming that FLASHATTENTION is as numerically stable as the baselines and produces the same training / validation curves.

Table 2: GPT-2 small and medium using FLASHATTENTION achieve up to 3 $\times$  speed up compared to Huggingface implementation and up to 1.7 $\times$  compared to Megatron-LM. Training time reported on 8xA100s GPUs.

Model implementations	OpenWebText (ppl)	Training time (speedup)
GPT-2 small - Huggingface [87]	18.2	9.5 days (1.0 $\times$ )
GPT-2 small - Megatron-LM [77]	18.2	4.7 days (2.0 $\times$ )
GPT-2 small - FLASHATTENTION	18.2	<b>2.7 days (3.5<math>\times</math>)</b>
GPT-2 medium - Huggingface [87]	14.2	21.0 days (1.0 $\times$ )
GPT-2 medium - Megatron-LM [77]	14.3	11.5 days (1.8 $\times$ )
GPT-2 medium - FLASHATTENTION	14.3	<b>6.9 days (3.0<math>\times</math>)</b>

**Long-range Arena.** We compare vanilla Transformer (with either standard implementation or FLASHATTENTION) on the long-range arena (LRA [80]) benchmark. We measure accuracy, throughput, and training time of all models. Each task has a different sequence length varying between 1024 and 4096. We follow the implementation and experimental setting in Tay et al. [80] and Xiong et al. [90].<sup>3</sup> Table 3 shows that FLASHATTENTION achieves up 2.4 $\times$  speed-up compared to standard attention. Block-sparse FLASHATTENTION is faster than all of the approximate attention methods that we have tested.

Table 3: The performance of standard attention, FLASHATTENTION, block-sparse FLASHATTENTION, and approximate attention baselines on the Long-Range-Arena benchmarks.

Models	ListOps	Text	Retrieval	Image	Pathfinder	Avg	Speedup
Transformer	36.0	63.6	81.6	42.3	72.7	59.3	-
FLASHATTENTION	37.6	63.9	81.4	43.5	72.7	59.8	2.4 $\times$
Block-sparse FLASHATTENTION	37.0	63.0	81.3	43.6	73.3	59.6	<b>2.8<math>\times</math></b>
Linformer [84]	35.6	55.9	77.7	37.8	67.6	54.9	2.5 $\times$
Linear Attention [50]	38.8	63.2	80.7	42.6	72.5	59.6	2.3 $\times$
Performer [12]	36.8	63.6	82.2	42.1	69.9	58.9	1.8 $\times$
Local Attention [80]	36.1	60.2	76.7	40.6	66.6	56.0	1.7 $\times$
Reformer [51]	36.5	63.8	78.5	39.6	69.4	57.6	1.3 $\times$
Smyrf [19]	36.1	64.1	79.0	39.6	70.5	57.9	1.7 $\times$

## 4.2 Better Models with Longer Sequences

**Language Modeling with Long Context.** The runtime and memory-efficiency of FLASHATTENTION allow us to increase the context length of GPT-2 by 4 $\times$  while still running faster than the optimized implementation from Megatron-LM. Table 4 shows that that GPT-2 with FLASHATTENTION and context length 4K is still 30% faster than GPT-2 from Megatron with context length 1K, while achieving 0.7 better perplexity.

Table 4: GPT-2 small with FLASHATTENTION, with 4 $\times$  larger context length compared to Megatron-LM, is still 30% faster while achieving 0.7 better perplexity. Training time on 8xA100 GPUs is reported.

Model implementations	Context length	OpenWebText (ppl)	Training time (speedup)
GPT-2 small - Megatron-LM	1k	18.2	4.7 days (1.0 $\times$ )
GPT-2 small - FLASHATTENTION	1k	18.2	<b>2.7 days (1.7<math>\times</math>)</b>
GPT-2 small - FLASHATTENTION	2k	17.6	3.0 days (1.6 $\times$ )
GPT-2 small - FLASHATTENTION	4k	<b>17.5</b>	3.6 days (1.3 $\times$ )

**Long Document Classification.** Training Transformers with longer sequences with FLASHATTENTION improves performance on the MIMIC-III [47] and ECtHR [6, 7] datasets. MIMIC-III contains intensive care unit patient discharge summaries, each annotated with multiple labels. ECtHR contains legal cases from the

<sup>3</sup>LRA accuracy results are known to be highly dependent on the tuning procedure [90]. Our reproduced baselines perform better than as reported in the original comparison [80].

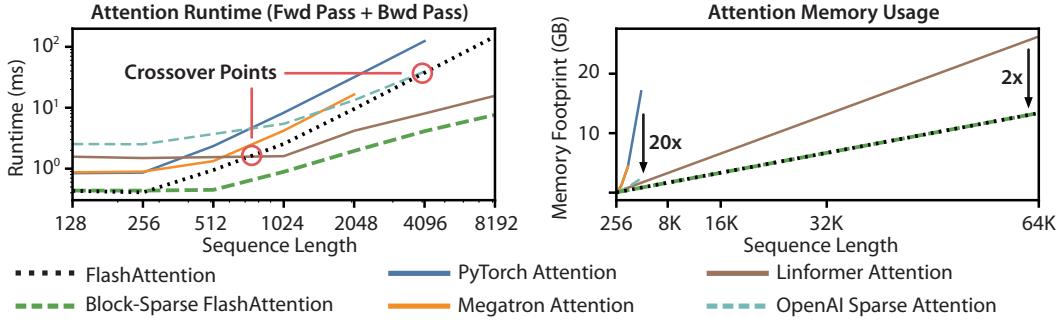


Figure 3: **Left:** runtime of forward pass + backward pass. **Right:** attention memory usage.

European Court of Human Rights, each of which is mapped to articles of the Convention of Human Rights that were allegedly violated. Both of these datasets contain very long text documents; the average number of tokens in MIMIC is 2,395 tokens, and the longest document contains 14,562 tokens, while the average and longest numbers in ECtHR are 2,197 and 49,392, respectively. We evaluate lift from increasing the sequence length of a pretrained RoBERTa model [56] (we repeat the positional embeddings, as in Beltagy et al. [3]).

Table 5 shows that sequence length 16K outperforms length 512 by 4.3 points on MIMIC, and that length 8K outperforms length 512 by 8.5 points on ECtHR. The discrepancies may be due to subtle distribution shifts: MIMIC-III contains specialized medical text and thus may be more susceptible to a distribution shift in the document length, whereas ECtHR contains general language.

Table 6: We report the first Transformer model that can achieve non-random performance on Path-X and Path-256.

Table 5: Long Document performance (micro  $F_1$ ) at different sequence lengths using FLASHATTENTION.

	512	1024	2048	4096	8192	16384
MIMIC-III [47]	52.8	50.7	51.7	54.6	56.4	<b>57.1</b>
ECtHR [6]	72.2	74.3	77.1	78.6	<b>80.7</b>	79.2

Model	Path-X	Path-256
Transformer	x	x
Linformer [84]	x	x
Linear Attention [50]	x	x
Performer [12]	x	x
Local Attention [80]	x	x
Reformer [51]	x	x
SMYRF [19]	x	x
FLASHATTENTION	<b>61.4</b>	x
Block-sparse FLASHATTENTION	56.0	<b>63.1</b>

**Path-X and Path-256.** The Path-X and Path-256 benchmarks are challenging tasks from the long-range arena benchmark designed to test long context. The task is to classify whether two points in a black and white 128×128 (or 256×256) image have a path connecting them, and the images are fed to the transformer one pixel at a time. In prior work, all transformer models have either run out of memory, or only achieved random performance [80]. There has been a search for alternative architectures that can model such long context [37]. We present here the first result of Transformer models being able to solve Path-X and Path-256 (Table 6). We pretrain a transformer on Path-64, and then transfer to Path-X by spatially interpolating the positional embeddings. FLASHATTENTION achieves 61.4 accuracy on Path-X. Additionally, block-sparse FLASHATTENTION enables the Transformers to scale to sequence length 64K, achieving 63.1 accuracy<sup>4</sup> on Path-256.

### 4.3 Benchmarking Attention

We vary sequence length and measure runtime and memory usage of FLASHATTENTION and block-sparse FLASHATTENTION against various attention baselines on one A100 GPU with 40 GB HBM, with dropout and a padding mask. We compare against reference implementations for exact attention, approximate attention, and sparse attention. We report a subset of baselines in the main body; Appendix E contains more baselines and full details.

<sup>4</sup>Path-256 requires longer sequences but has relatively shorter paths than Path-X, so it is easier to obtain a higher accuracy.

**Runtime.** Figure 3 (left) reports the runtime in milliseconds of the forward + backward pass of FLASHATTENTION and block-sparse FLASHATTENTION compared to the baselines in exact, approximate, and sparse attention (exact numbers in Appendix E). Runtime grows quadratically with sequence length, but FLASHATTENTION runs significantly faster than **exact attention** baselines, up to 3 $\times$  faster than the PyTorch implementation. The runtimes of many approximate/sparse attention mechanisms grow linearly with sequence length, but FLASHATTENTION still runs faster than approximate and sparse attention for short sequences due to fewer memory accesses. The **approximate attention** runtimes begin to cross over with FLASHATTENTION at sequences between 512 and 1024. On the other hand, block-sparse FLASHATTENTION is faster than all implementations of exact, sparse, and approximate attention that we know of, across all sequence lengths.

**Memory Footprint.** Figure 3 (right) shows the memory footprint of FLASHATTENTION and block-sparse FLASHATTENTION compared to various exact, approximate, and sparse attention baselines. FLASHATTENTION and block-sparse FLASHATTENTION have the same memory footprint, which grows linearly with sequence length. FLASHATTENTION is up to 20 $\times$  more memory efficient than **exact attention** baselines, and is more memory-efficient than the **approximate attention** baselines. All other algorithms except for Linformer run out of memory on an A100 GPU before 64K, and FLASHATTENTION is still 2 $\times$  more efficient than Linformer.

## 5 Limitations and Future Directions

We discuss limitations of our approach and future directions. Related work is given in Appendix A.

**Compiling to CUDA.** Our current approach to building IO-aware implementations of attention requires writing a new CUDA kernel for each new attention implementation. This requires writing the attention algorithm in a considerably lower-level language than PyTorch, and requires significant engineering effort. Implementations may also not be transferrable across GPU architectures. These limitations suggest the need for a method that supports writing attention algorithms in a high-level language (e.g., PyTorch), and compiling to IO-aware implementations in CUDA—similar to efforts such as Halide in image processing [70].

**IO-Aware Deep Learning.** We believe that the IO-aware approach can extend beyond attention. Attention is the most memory-intensive computation in Transformers, but every layer in a deep network touches GPU HBM. We hope our work inspires IO-aware implementations of additional modules. We discuss these potential extensions in Appendix D.

**Multi-GPU IO-Aware Methods.** Our IO-aware implementation of attention is optimal within constants for computing attention on a single GPU. However, the attention computation may be parallelizable across multiple GPUs [72]. Using multiple GPUs adds an additional layer to IO analysis—accounting for data transfer between GPUs. We hope our work inspires future work in this direction.

### Acknowledgments

Our implementation uses Apex’s FMHA code (<https://github.com/NVIDIA/apex/tree/master/apex/contrib/csrc/fmha>) as a starting point. We thank Young-Jun Ko for the in-depth explanation of his FMHA implementation and for his thoughtful answers to our questions about CUDA. We thank Sabri Eyuboglu, Megan Leszczynski, Laurel Orr, Yuhuai Wu, Beidi Chen, and Xun Huang for their constructive feedback and suggestions on early drafts of the paper. We thank Markus Rabe and Charles Staats for helpful discussion of their attention algorithm.

We gratefully acknowledge the support of NIH under No. U54EB020405 (Mobilize), NSF under Nos. CCF1763315 (Beyond Sparsity), CCF1563078 (Volume to Velocity), and 1937301 (RTML); ARL under No. W911NF-21-2-0251 (Interactive Human-AI Teaming); ONR under No. N000141712266 (Unifying Weak Supervision); ONR N00014-20-1-2480: Understanding and Applying Non-Euclidean Geometry in Machine Learning; N000142012275 (NEPTUNE); NXP, Xilinx, LETI-CEA, Intel, IBM, Microsoft, NEC, Toshiba, TSMC, ARM, Hitachi, BASF, Accenture, Ericsson, Qualcomm, Analog Devices, Google Cloud, Salesforce, Total, the HAI-GCP & HAI-Azure Cloud Credits for Research program, the Stanford Data Science Initiative (SDSI), Department of Defense (DoD) through the National Defense Science and Engineering Graduate Fellowship (NDSEG) Program, and members of the Stanford DAWN project: Facebook, Google, and VMWare. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes

notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of NIH, ONR, or the U.S. Government. Atri Rudra's research is supported by NSF grant CCF-1763481.

## References

- [1] Alok Aggarwal and S Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
- [2] Irwan Bello. LambdaNetworks: Modeling long-range interactions without attention. *arXiv preprint arXiv:2102.08602*, 2021.
- [3] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.
- [4] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.
- [5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- [6] Ilias Chalkidis, Ion Androutsopoulos, and Nikolaos Aletras. Neural legal judgment prediction in English. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4317–4323, Florence, Italy, 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1424. URL <https://www.aclweb.org/anthology/P19-1424>.
- [7] Ilias Chalkidis, Manos Fergadiotis, Dimitrios Tsarapatsanis, Nikolaos Aletras, Ion Androutsopoulos, and Prodromos Malakasiotis. Paragraph-level rationale extraction through regularization: A case study on european court of human rights cases. In *Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics*, Mexico City, Mexico, 2021. Association for Computational Linguistics.
- [8] Benjamin Charlier, Jean Feydy, Joan Alexis Glaunès, François-David Collin, and Ghislain Durif. Kernel operations on the gpu, with autodiff, without memory overflows. *Journal of Machine Learning Research*, 22(74):1–6, 2021. URL <http://jmlr.org/papers/v22/20-275.html>.
- [9] Beidi Chen, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra, and Christopher Ré. Scatterbrain: Unifying sparse and low-rank attention. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.
- [10] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.
- [11] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.
- [12] Krzysztof Marcin Choromanski, Valerii Likhoshesterov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. In *International Conference on Learning Representations (ICLR)*, 2020.
- [13] Xiang Dai, Ilias Chalkidis, Sune Darkner, and Desmond Elliott. Revisiting transformer-based models for long document classification. *arXiv preprint arXiv:2204.06683*, 2022.
- [14] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, 2019.

- [15] Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Ré. Learning fast algorithms for linear transforms using butterfly factorizations. In *International Conference on Machine Learning (ICML)*, 2019.
- [16] Tri Dao, Nimit Sohoni, Albert Gu, Matthew Eichhorn, Amit Blonder, Megan Leszczynski, Atri Rudra, and Christopher Ré. Kaleidoscope: An efficient, learnable representation for all structured linear maps. In *International Conference on Learning Representations (ICLR)*, 2020.
- [17] Tri Dao, Beidi Chen, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Ré. Pixelated butterfly: Simple and efficient sparse training for neural network models. In *International Conference on Learning Representations (ICLR)*, 2022.
- [18] Tri Dao, Beidi Chen, Nimit Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning (ICML)*, 2022.
- [19] Giannis Daras, Nikita Kitaev, Augustus Odena, and Alexandros G Dimakis. Smyrf-efficient attention using asymmetric clustering. *Advances in Neural Information Processing Systems*, 33:6476–6489, 2020.
- [20] Christopher De Sa, Albert Gu, Rohan Puttagunta, Christopher Ré, and Atri Rudra. A two-pronged progress in structured dense matrix vector multiplication. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1060–1079. SIAM, 2018.
- [21] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.
- [22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. 2019.
- [23] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. *arXiv preprint arXiv:1705.07565*, 2017.
- [24] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2020.
- [25] Y Eidelman and I Gohberg. On a new class of structured matrices. *Integral Equations and Operator Theory*, 34(3):293–324, 1999.
- [26] Jean Feydy, Joan Glaunès, Benjamin Charlier, and Michael Bronstein. Fast geometric learning with symbolic matrices. *Advances in Neural Information Processing Systems*, 33, 2020.
- [27] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Springer, 2006.
- [28] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2018.
- [29] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M Roy, and Michael Carbin. Stabilizing the lottery ticket hypothesis. *arXiv preprint arXiv:1903.01611*, 2019.
- [30] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel Roy, and Michael Carbin. Linear mode connectivity and the lottery ticket hypothesis. In *International Conference on Machine Learning*, pages 3259–3269. PMLR, 2020.
- [31] Karan Goel, Albert Gu, Chris Donahue, and Christopher Ré. It’s raw! audio generation with state-space models. In *International Conference on Machine Learning (ICML)*, 2022.
- [32] Aaron Gokaslan, Vanya Cohen, Pavlick Ellie, and Stefanie Tellex. Openwebtext corpus, 2019.

- [33] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1(1):29–53, 1997.
- [34] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.
- [35] Albert Gu, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Ré. Hippo: Recurrent memory with optimal polynomial projections. In *Advances in neural information processing systems (NeurIPS)*, 2020.
- [36] Albert Gu, Isys Johnson, Karan Goel, Khaled Saab, Tri Dao, Atri Rudra, and Christopher Ré. Combining recurrent, convolutional, and continuous-time models with linear state space layers. *Advances in Neural Information Processing Systems*, 34, 2021.
- [37] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. In *The International Conference on Learning Representations (ICLR)*, 2022.
- [38] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015.
- [39] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations*, 2016.
- [40] John Hennessy and David Patterson. Memory hierarchy design. *Computer Architecture: A Quantitative Approach*, pages 390–525, 2003.
- [41] Sara Hooker. The hardware lottery. *arXiv preprint arXiv:2009.06489*, 2020.
- [42] Weizhe Hua, Zihang Dai, Hanxiao Liu, and Quoc V Le. Transformer quality in linear time. *arXiv preprint arXiv:2202.10447*, 2022.
- [43] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3: 711–732, 2021.
- [44] Zhe Jia and Peter Van Sandt. Dissecting the Ampere GPU architecture via microbenchmarking. GPU Technology Conference, 2021.
- [45] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia Volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.
- [46] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore IPU architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413*, 2019.
- [47] Alistair EW Johnson, Tom J Pollard, Lu Shen, Li-wei H Lehman, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G Mark. Mimic-iii, a freely accessible critical care database. *Scientific data*, 3(1):1–9, 2016.
- [48] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.
- [49] Thomas Kailath, Sun-Yuan Kung, and Martin Morf. Displacement ranks of matrices and linear equations. *Journal of Mathematical Analysis and Applications*, 68(2):395–407, 1979.
- [50] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, pages 5156–5165. PMLR, 2020.

- [51] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *The International Conference on Machine Learning (ICML)*, 2020.
- [52] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite BEDRT for self-supervised learning of language representations. In *The International Conference on Learning Representations (ICLR)*, 2020.
- [53] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.
- [54] Valerii Likhoshesterov, Krzysztof Choromanski, Jared Davis, Xingyou Song, and Adrian Weller. Sub-linear memory: How to make performers slim. *arXiv preprint arXiv:2012.11346*, 2020.
- [55] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime neural pruning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.
- [56] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.
- [57] Xuezhe Ma, Xiang Kong, Sinong Wang, Chunting Zhou, Jonathan May, Hao Ma, and Luke Zettlemoyer. Luna: Linear unified nested attention. *Advances in Neural Information Processing Systems*, 34, 2021.
- [58] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bitdorf, et al. Mlperf training benchmark. *Proceedings of Machine Learning and Systems*, 2:336–349, 2020.
- [59] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what {COST}? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.
- [60] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.
- [61] NVIDIA. Nvidia Tesla V100 GPU architecture, 2017.
- [62] NVIDIA. Nvidia A100 tensor core GPU architecture, 2020.
- [63] NVIDIA. Nvidia H100 tensor core GPU architecture, 2022.
- [64] D Stott Parker. Random butterfly transformations with applications in computational linear algebra. 1995.
- [65] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.
- [66] Markus N Rabe and Charles Staats. Self-attention does not need  $O(n^2)$  memory. *arXiv preprint arXiv:2112.05682*, 2021.
- [67] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- [68] Jack Rae and Ali Razavi. Do transformers need deep long-range memory? In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Online, July 2020. Association for Computational Linguistics. URL <https://www.aclweb.org/anthology/2020.acl-main.672>.
- [69] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. In *The International Conference on Learning Representations (ICLR)*, 2020.

- [70] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.
- [71] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003.
- [72] Benjamin Recht and Christopher Ré. Parallel stochastic gradient algorithms for large-scale matrix completion. *Mathematical Programming Computation*, 5(2):201–226, 2013.
- [73] Hongyu Ren, Hanjun Dai, Zihang Dai, Mengjiao Yang, Jure Leskovec, Dale Schuurmans, and Bo Dai. Combiner: Full attention transformer with sparse computation cost. *Advances in Neural Information Processing Systems*, 34, 2021.
- [74] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9: 53–68, 2021.
- [75] Amit Sabne. XLA: Compiling machine learning for peak performance. 2020.
- [76] Victor Sanh, Thomas Wolf, and Alexander M Rush. Movement pruning: Adaptive sparsity by fine-tuning. *arXiv preprint arXiv:2005.07683*, 2020.
- [77] Mohammad Shoeybi, Mostafa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [78] Vikas Sindhwani, Tara Sainath, and Sanjiv Kumar. Structured transforms for small-footprint deep learning. In *Advances in Neural Information Processing Systems*, pages 3088–3096, 2015.
- [79] Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2019.
- [80] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. In *International Conference on Learning Representations*, 2020.
- [81] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*, 2020.
- [82] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [83] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Dongdong Zhang, and Furu Wei. Deepnet: Scaling transformers to 1,000 layers. *arXiv preprint arXiv:2203.00555*, 2022.
- [84] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.
- [85] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.
- [86] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, 1991.

- [87] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierrick Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. URL <https://www.aclweb.org/anthology/2020.emnlp-demos.6>.
- [88] David P Woodruff. Optimal space lower bounds for all frequency moments. In *SODA*, volume 4, pages 167–175. Citeseer, 2004.
- [89] Felix Wu, Angela Fan, Alexei Baevski, Yann N Dauphin, and Michael Auli. Pay less attention with lightweight and dynamic convolutions. In *The International Conference on Learning Representations (ICLR)*, 2019.
- [90] Yunyang Xiong, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, and Vikas Singh. Nyströmformer: A nyström-based algorithm for approximating self-attention. In *Proceedings of the AAAI Conference on Artificial Intelligence. AAAI Conference on Artificial Intelligence*, volume 35, page 14138, 2021.
- [91] Li Yuan, Yunpeng Chen, Tao Wang, Weihao Yu, Yujun Shi, Zi-Hang Jiang, Francis EH Tay, Jiashi Feng, and Shuicheng Yan. Tokens-to-token vit: Training vision transformers from scratch on imagenet. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 558–567, 2021.
- [92] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in Neural Information Processing Systems*, 33, 2020.
- [93] Shuangfei Zhai, Walter Talbott, Nitish Srivastava, Chen Huang, Hanlin Goh, Ruixiang Zhang, and Josh Susskind. An attention free transformer. *arXiv preprint arXiv:2105.14103*, 2021.
- [94] Chen Zhu, Wei Ping, Chaowei Xiao, Mohammad Shoeybi, Tom Goldstein, Anima Anandkumar, and Bryan Catanzaro. Long-short transformer: Efficient transformers for language and vision. *Advances in Neural Information Processing Systems*, 34, 2021.

## A Related Work

**IO-Aware Runtime Optimization.** The broad concept of optimizing for reading and writing to fast/slow memory has a long history in computer science and has been known by many names. We draw the most direct connection to the literature of analyzing I/O complexity in this work [1], but concepts of memory hierarchies are fundamental and have appeared in many forms, from the working set model [21], to data locality [86], to the Roofline model of arithmetic intensity [85], to analyses of scalability [59], to standard textbook treatments of computer architecture [40]. We hope that this work encourages the community to adopt these ideas in more parts of the deep learning stack.

**Efficient ML Models with Structured Matrices.** Matrix multiply is the core computational bottleneck of most machine learning models. To reduce the computational complexity, there have been numerous approaches to learn over a more efficient set of matrices. These matrices are called *structured matrices*, which have subquadratic ( $o(n^2)$ ) for dimension  $n \times n$ ) number of parameters and runtime. Most common examples of structured matrices are sparse and low-rank matrices, along with fast transforms commonly encountered in signal processing (Fourier, Chebyshev, sine/cosine, orthogonal polynomials). There have been several more general classes of structured matrices proposed in machine learning: Toeplitz-like [78], low-displacement rank [49], quasi-separable [25]). The butterfly pattern we use for our block-sparse attention is motivated by the fact that butterfly matrices [15, 64] and their products have been shown to be able to express any structured matrices with almost optimal runtime and number of parameters [16, 20]. However, even though structured matrices are efficient in theory, they have not seen wide adoption since it is hard to translate their efficiency to wall-clock speedup since dense unconstrained matrix multiply has very optimize implementation, a phenomenon known as the hardware lottery [41]. Extensions of butterfly matrices [17, 18] aimed to make butterfly matrices more hardware-friendly.

**Sparse Training.** Our block-sparse FLASHATTENTION can be seen as a step towards making sparse model training more efficient. Sparse models have seen success in compressing models for inference (pruning) by sparsifying the weight matrices [23, 38, 39, 55, 76]. For model training, the lottery tickets hypothesis [28, 29, 30] suggests that there are a set of small sub-networks derived from a larger dense network that performs as well as the original dense network. Out block-sparse FLASHATTENTION can also be seen as a fixed lottery ticket in the context of attention: we fix the sparsity pattern to be the butterfly pattern through training, and observe that it performs almost as well as the (dense) FLASHATTENTION on the Long-range Arena tasks.

**Efficient Transformer.** Transformer-based models have become the most widely-used architecture in natural language processing [22] and computer vision [24, 91]. However, one of their computational bottlenecks is that their time and memory scales quadratic in the sequence length. There are numerous approaches to overcome this bottleneck, including approximation with hashing (i.e., sparse) such as Reformer [51] and Smyrf [19] and with low-rank approximation such as Performer [12, 54]. One can even combine sparse and low-rank approximation for better accuracy (e.g., Longformer [3], BigBird [92], Scatterbrain [9], Long-short transformer [94], Combiner [73]). Other approaches include compressing along the sequence dimension to attend to multiple tokens at once [52, 57, 79, 89]. One can also attend over the states from previous sequences to help lengthen the context (e.g., Transformer-XL [14] and Compressive Transformer [69]). We recommend the survey [81] for more details.

There are several lines of work on developing other modules instead of attention to model longer context. HiPPO [35] and its extensions, most notably S4 [31, 36, 37] projects the history on a polynomial basis, allowing accurate reconstruction of the history through state-space models. They combine the strengths of CNNs (efficient training), RNNs (efficient inference), and continuous models (robust to change in sampling rates). LambdaNetworks [2], AFT [93] and FLASH [42] are other attempts at replacing attention in the context of image classification and language modeling.

## B Algorithm Details

We first derive the forward and backward passes of attention and show that they can be computed in a memory-efficient manner (requiring extra memory linear instead of quadratic in the sequence length). Though they reduce the amount of extra memory required, naively they still incur quadratic HBM accesses, resulting in slower execution speed. We describe the FLASHATTENTION algorithm to implement both the forward

and the backward passes on GPUs that reduces HBM accesses, leading to both faster runtime and smaller memory footprint.

## B.1 Memory-efficient forward pass

The main challenge in making attention memory-efficient is the softmax that couples the columns of  $\mathbf{K}$  (and columns of  $\mathbf{V}$ ). Our approach is to compute the softmax normalization constant separately to decouple the columns. This technique [60] has been used in the literature [51, 66] to show that attention computation does not need quadratic *extra* memory (though the number of HBM accesses is still quadratic, resulting in slow run-time).

For simplicity, we omit here the max-shifting step during softmax. The full algorithm in Appendix B.3 contains all the steps.

Recall that given input sequences  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ , we want to compute the attention output  $\mathbf{O} \in \mathbb{R}^{N \times d}$ :

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \text{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d}.$$

We have that  $S_{ij} = q_i^T k_j$  where  $q_i$  and  $k_j$  are the  $i$ -th and  $j$ -th columns of  $\mathbf{Q}$  and  $\mathbf{K}$  respectively. Define the normalization constants of softmax:

$$L_i = \sum_j e^{q_i^T k_j}. \quad (1)$$

Let  $v_j$  be the  $j$ -th column of  $\mathbf{V}$ , then the  $i$ -th columns of the output is

$$o_i = P_{i:}\mathbf{V} = \sum_j P_{ij}v_j = \sum_j \frac{e^{q_i^T k_j}}{L_i} v_j. \quad (2)$$

We see that once  $L_i$  is computed, we can compute  $o_i$  without extra memory by repeatedly summing  $\frac{e^{q_i^T k_j}}{L_i} v_j$ . Therefore the forward pass can be computed with  $O(n)$  extra memory:

1. Compute  $L_i$  for all  $i$  according to Eq. (1), which takes  $O(n)$  extra memory.
2. Compute  $o_i$  for all  $i$  according to Eq. (2), which takes  $O(d)$  extra memory.

## B.2 Memory-efficient backward pass

We derive the backward pass of attention and show that it can also be computed with linear memory. Rabe and Staats [66] suggests that the backward pass can be done without quadratic extra memory by applying gradient checkpointing to the memory-efficient forward pass. We instead derive the backward pass explicitly and show how it can be computed in a memory-efficient manner.

Suppose that there is a scalar loss function  $\phi$ , and let the output gradient be  $\mathbf{dO} \in \mathbb{R}^{n \times d}$  (where  $\mathbf{dO}$  denotes  $\frac{\partial \phi}{\partial \mathbf{O}}$ ). We want to compute the input gradients  $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV} \in \mathbb{R}^{n \times d}$  (where  $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$  denote  $\frac{\partial \phi}{\partial \mathbf{Q}}, \frac{\partial \phi}{\partial \mathbf{K}}, \frac{\partial \phi}{\partial \mathbf{V}}$  respectively).

The gradient  $\mathbf{dV}$  is easy to see. Applying reverse-mode autodiff by hand (aka the chain rule), we obtain (in matrix notation)  $\mathbf{dV} = \mathbf{P}^T \mathbf{dO}$ . Thus:

$$dv_j = \sum_i P_{ij} do_i = \sum_i \frac{e^{q_i^T k_j}}{L_i} do_i. \quad (3)$$

Since we already computed  $L_i$ ,  $dv_j$  can be computed without extra memory by repeated summing.

The gradients  $\mathbf{dQ}$  and  $\mathbf{dK}$  are a little more complicated. We go through the gradients  $\mathbf{dP}$  and  $\mathbf{dS}$  first. From Eq. (2), we have that  $\mathbf{dP} = \mathbf{dO}\mathbf{V}^T$ , and so:

$$dP_{ij} = do_i^T v_j.$$

Recall that  $P_{i:} = \text{softmax}(S_{i:})$ . Using the fact that the Jacobian of  $y = \text{softmax}(x)$  is  $\text{diag}(y) - yy^T$ , we have that

$$dS_{i:} = (\text{diag}(P_{i:}) - P_{i:}P_{i:}^T)dP_{i:} = P_{i:} \circ dP_{i:} - (P_{i:}^T dP_{i:})P_{i:},$$

where  $\circ$  denotes pointwise multiplication.

Define

$$D_i = P_{i:}^T dP_{i:} = \sum_j \frac{e^{q_i^T k_j}}{L_i} do_i^T v_j = do_i^T \sum_j \frac{e^{q_i^T k_j}}{L_i} v_j = do_i^T o_i, \quad (4)$$

then

$$dS_{i:} = P_{i:} \circ dP_{i:} - D_i P_{i:}.$$

Hence

$$dS_{ij} = P_{ij} dP_{ij} - D_i P_{ij} = P_{ij} (dP_{ij} - D_i).$$

Now we can get the gradients  $\mathbf{dQ}$  and  $\mathbf{dK}$ . Recall that  $S_{ij} = q_i^T k_j$ , so

$$dq_i = \sum_j dS_{ij} k_j = \sum_j P_{ij} (dP_{ij} - D_i) k_j = \sum_j \frac{e^{q_i^T k_j}}{L_i} (do_i^T v_j - D_i) k_j. \quad (5)$$

Similarly,

$$dk_j = \sum_i dS_{ij} q_i = \sum_i P_{ij} (dP_{ij} - D_i) q_i = \sum_i \frac{e^{q_i^T k_j}}{L_i} (do_i^T v_j - D_i) q_i. \quad (6)$$

Therefore the backward pass can also be computed with  $O(n)$  extra memory:

1. Compute  $dv_j$  for all  $j$  according to Eq. (3), which takes  $O(d)$  extra memory.
2. Compute  $D_i$  for all  $i$  according to Eq. (4), which takes  $O(n)$  extra memory.
3. Compute  $dq_i$  for all  $i$  according to Eq. (5), which takes  $O(d)$  extra memory.
4. Compute  $dk_j$  for all  $j$  according to Eq. (6), which takes  $O(d)$  extra memory.

### B.3 FLASHATTENTION: Forward Pass

We describe the full details of FLASHATTENTION forward pass. Given input sequences  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ , we want to compute the attention output  $\mathbf{O} \in \mathbb{R}^{N \times d}$ :

$$\begin{aligned} \mathbf{S} &= \tau \mathbf{Q} \mathbf{K}^\top \in \mathbb{R}^{N \times N}, \quad \mathbf{S}^{\text{masked}} = \text{MASK}(\mathbf{S}) \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \text{softmax}(\mathbf{S}^{\text{masked}}) \in \mathbb{R}^{N \times N}, \\ \mathbf{P}^{\text{dropped}} &= \text{dropout}(\mathbf{P}, p_{\text{drop}}), \quad \mathbf{O} = \mathbf{P}^{\text{dropped}} \mathbf{V} \in \mathbb{R}^{N \times d}, \end{aligned}$$

where  $\tau \in \mathbb{R}$  is some softmax scaling (typically  $\frac{1}{\sqrt{d}}$ ), MASK is some masking function that sets some entries of the input to  $-\infty$  and keep other entries the same (e.g., key padding mask when sequences in the batch don't have the same lengths and are padded), and  $\text{dropout}(x, p)$  applies dropout to  $x$  elementwise (i.e., output  $\frac{x}{1-p}$  with probability  $1-p$  and output 0 with probability  $p$  for each element  $x$ ).

The full algorithm is in Algorithm 2. We save the output  $\mathbf{O}$ , the softmax statistics  $\ell$  and  $m$ , and the pseudo-random number generator state  $\mathcal{R}$  for the backward pass.

---

**Algorithm 2** FLASHATTENTION Forward Pass

---

**Require:** Matrices  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$  in HBM, on-chip SRAM of size  $M$ , softmax scaling constant  $\tau \in \mathbb{R}$ , masking function MASK, dropout probability  $p_{\text{drop}}$ .

- 1: Initialize the pseudo-random number generator state  $\mathcal{R}$  and save to HBM.
- 2: Set block sizes  $B_c = \lceil \frac{M}{4d} \rceil$ ,  $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$ .
- 3: Initialize  $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$ ,  $\ell = (0)_N \in \mathbb{R}^N$ ,  $m = (-\infty)_N \in \mathbb{R}^N$  in HBM.
- 4: Divide  $\mathbf{Q}$  into  $T_r = \lceil \frac{N}{B_r} \rceil$  blocks  $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$  of size  $B_r \times d$  each, and divide  $\mathbf{K}, \mathbf{V}$  in to  $T_c = \lceil \frac{N}{B_c} \rceil$  blocks  $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$  and  $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$ , of size  $B_c \times d$  each.
- 5: Divide  $\mathbf{O}$  into  $T_r$  blocks  $\mathbf{O}_1, \dots, \mathbf{O}_{T_r}$  of size  $B_r \times d$  each, divide  $\ell$  into  $T_r$  blocks  $\ell_1, \dots, \ell_{T_r}$  of size  $B_r$  each, divide  $m$  into  $T_r$  blocks  $m_1, \dots, m_{T_r}$  of size  $B_r$  each.
- 6: **for**  $1 \leq j \leq T_c$  **do**
- 7:   Load  $\mathbf{K}_j, \mathbf{V}_j$  from HBM to on-chip SRAM.
- 8:   **for**  $1 \leq i \leq T_r$  **do**
- 9:     Load  $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$  from HBM to on-chip SRAM.
- 10:    On chip, compute  $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$ .
- 11:    On chip, compute  $\mathbf{S}_{ij}^{\text{masked}} = \text{MASK}(\mathbf{S}_{ij})$ .
- 12:    On chip, compute  $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}^{\text{masked}}) \in \mathbb{R}^{B_r}$ ,  $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij}^{\text{masked}} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$  (pointwise),  $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$ .
- 13:    On chip, compute  $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$ ,  $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$ .
- 14:    On chip, compute  $\tilde{\mathbf{P}}_{ij}^{\text{dropped}} = \text{dropout}(\tilde{\mathbf{P}}_{ij}, p_{\text{drop}})$ .
- 15:    Write  $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij}^{\text{dropped}} \mathbf{V}_j)$  to HBM.
- 16:    Write  $\ell_i \leftarrow \ell_i^{\text{new}}$ ,  $m_i \leftarrow m_i^{\text{new}}$  to HBM.
- 17:   **end for**
- 18: **end for**
- 19: Return  $\mathbf{O}, \ell, m, \mathcal{R}$ .

---

## B.4 FLASHATTENTION: Backward Pass

We describe the full details of FLASHATTENTION backward pass. Given input sequences  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ , the output  $\mathbf{O} \in \mathbb{R}^{N \times d}$ , and the output gradient  $\mathbf{dO}$ , we want to compute the input gradients  $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV} \in \mathbb{R}^{N \times d}$ .

We first describe the standard attention backward pass in Algorithm 3 for completeness.

---

**Algorithm 3** Standard Attention Backward Pass

---

**Require:** Matrices  $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{dO} \in \mathbb{R}^{N \times d}$ ,  $\mathbf{P} \in \mathbb{R}^{N \times N}$  in HBM.

- 1: Load  $\mathbf{P}, \mathbf{dO}$  by blocks from HBM, compute  $\mathbf{dV} = \mathbf{P}^T \mathbf{dO} \in \mathbb{R}^{N \times d}$ , write  $\mathbf{dV}$  to HBM.
  - 2: Load  $\mathbf{dO}, \mathbf{V}$  by blocks from HBM, compute  $\mathbf{dP} = \mathbf{dO} \mathbf{V}^T \in \mathbb{R}^{N \times N}$ , write  $\mathbf{dP}$  to HBM.
  - 3: Read  $\mathbf{P}, \mathbf{dP}$  from HBM, compute  $\mathbf{dS} \in \mathbb{R}^{N \times N}$  where  $dS_{ij} = P_{ij} (dP_{ij} - \sum_l P_{il} dP_{il})$ , write  $\mathbf{dS}$  to HBM.
  - 4: Load  $\mathbf{dS}$  and  $\mathbf{K}$  by blocks from HBM, compute  $\mathbf{dQ} = \mathbf{dSK}$ , write  $\mathbf{dQ}$  to HBM.
  - 5: Load  $\mathbf{dS}$  and  $\mathbf{Q}$  by blocks from HBM, compute  $\mathbf{dK} = \mathbf{dS}^T \mathbf{Q}$ , write  $\mathbf{dK}$  to HBM.
  - 6: Return  $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$ .
- 

We now make two observations about FLASHATTENTION backward pass:

1. We do not need to store the dropout mask of size  $O(N^2)$  from the forward pass. Instead, we can save the pseudo-random number generator states from the forward pass and re-generate the dropout mask in the backward pass. This allows us to only use  $O(N)$  extra memory.
2. When computing the softmax gradient, we use Eq. (4) to compute  $D_i = P_{i:}^T dP_{i:}$  without reducing over  $P_{i:}$  and  $dP_{i:}$  of size  $N$  (they might not fit into SRAM). Instead we can rewrite  $D_i = d\mathbf{o}_i^T \mathbf{o}_i$  and compute the dot product between vectors of size  $d$ .

The full FLASHATTENTION backward pass algorithm is in Algorithm 4. Conceptually it is just a block version of the derivation in Appendix B.2.

---

**Algorithm 4** FLASHATTENTION Backward Pass

---

**Require:** Matrices  $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}, \mathbf{dO} \in \mathbb{R}^{N \times d}$  in HBM, vectors  $\ell, m \in \mathbb{R}^N$  in HBM, on-chip SRAM of size  $M$ , softmax scaling constant  $\tau \in \mathbb{R}$ , masking function MASK, dropout probability  $p_{\text{drop}}$ , pseudo-random number generator state  $\mathcal{R}$  from the forward pass.

- 1: Set the pseudo-random number generator state to  $\mathcal{R}$ .
- 2: Set block sizes  $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$ .
- 3: Divide  $\mathbf{Q}$  into  $T_r = \lceil \frac{N}{B_r} \rceil$  blocks  $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$  of size  $B_r \times d$  each, and divide  $\mathbf{K}, \mathbf{V}$  in to  $T_c = \lceil \frac{N}{B_c} \rceil$  blocks  $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$  and  $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$ , of size  $B_c \times d$  each.
- 4: Divide  $\mathbf{O}$  into  $T_r$  blocks  $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$  of size  $B_r \times d$  each, divide  $\mathbf{dO}$  into  $T_r$  blocks  $\mathbf{dO}_i, \dots, \mathbf{dO}_{T_r}$  of size  $B_r \times d$  each, divide  $\ell$  into  $T_r$  blocks  $\ell_i, \dots, \ell_{T_r}$  of size  $B_r$  each, divide  $m$  into  $T_r$  blocks  $m_1, \dots, m_{T_r}$  of size  $B_r$  each.
- 5: Initialize  $\mathbf{dQ} = (0)_{N \times d}$  in HBM and divide it into  $T_r$  blocks  $\mathbf{dQ}_1, \dots, \mathbf{dQ}_{T_r}$  of size  $B_r \times d$  each. Initialize  $\mathbf{dK} = (0)_{N \times d}, \mathbf{dV} = (0)_{N \times d}$  in HBM and divide  $\mathbf{dK}, \mathbf{dV}$  in to  $T_c$  blocks  $\mathbf{dK}_1, \dots, \mathbf{dK}_{T_c}$  and  $\mathbf{dV}_1, \dots, \mathbf{dV}_{T_c}$ , of size  $B_c \times d$  each.
- 6: **for**  $1 \leq j \leq T_c$  **do**
- 7:   Load  $\mathbf{K}_j, \mathbf{V}_j$  from HBM to on-chip SRAM.
- 8:   Initialize  $\tilde{\mathbf{dK}}_j = (0)_{B_c \times d}, \tilde{\mathbf{dV}}_j = (0)_{B_c \times d}$  on SRAM.
- 9:   **for**  $1 \leq i \leq T_r$  **do**
- 10:     Load  $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, \ell_i, m_i$  from HBM to on-chip SRAM.
- 11:     On chip, compute  $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$ .
- 12:     On chip, compute  $\mathbf{S}_{ij}^{\text{masked}} = \text{MASK}(\mathbf{S}_{ij})$ .
- 13:     On chip, compute  $\mathbf{P}_{ij} = \text{diag}(\ell_i)^{-1} \exp(\mathbf{S}_{ij}^{\text{masked}} - m_i) \in \mathbb{R}^{B_r \times B_c}$ .
- 14:     On chip, compute dropout mask  $\mathbf{Z}_{ij} \in \mathbb{R}^{B_r \times B_c}$  where each entry has value  $\frac{1}{1-p_{\text{drop}}}$  with probability  $1 - p_{\text{drop}}$  and value 0 with probability  $p_{\text{drop}}$ .
- 15:     On chip, compute  $\mathbf{P}_{ij}^{\text{dropped}} = \mathbf{P}_{ij} \circ \mathbf{Z}_{ij}$  (pointwise multiply).
- 16:     On chip, compute  $\tilde{\mathbf{dV}}_j \leftarrow \tilde{\mathbf{dV}}_j + (\mathbf{P}_{ij}^{\text{dropped}})^T \mathbf{dO}_i \in \mathbb{R}^{B_c \times d}$ .
- 17:     On chip, compute  $\mathbf{dP}_{ij}^{\text{dropped}} = \mathbf{dO}_i \mathbf{V}_j^T \in \mathbb{R}^{B_r \times B_c}$ .
- 18:     On chip, compute  $\mathbf{dP}_{ij} = \mathbf{dP}_{ij}^{\text{dropped}} \circ \mathbf{Z}_{ij}$  (pointwise multiply).
- 19:     On chip, compute  $D_i = \text{rowsum}(\mathbf{dO}_i \circ \mathbf{O}_i) \in \mathbb{R}^{B_r}$ .
- 20:     On chip, compute  $\mathbf{dS}_{ij} = \mathbf{P}_{ij} \circ (\mathbf{dP}_{ij} - D_i) \in \mathbb{R}^{B_r \times B_c}$ .
- 21:     Write  $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \tau \mathbf{dS}_{ij} \mathbf{K}_j \in \mathbb{R}^{B_r \times d}$  to HBM.
- 22:     On chip, compute  $\tilde{\mathbf{dK}}_j \leftarrow \tilde{\mathbf{dK}}_j + \tau \mathbf{dS}_{ij}^T \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$ .
- 23:   **end for**
- 24:   Write  $\mathbf{dK}_j \leftarrow \tilde{\mathbf{dK}}_j, \mathbf{dV}_j \leftarrow \tilde{\mathbf{dV}}_j$  to HBM.
- 25: **end for**
- 26: Return  $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$ .

---

We see that similar to the forward pass, the backward pass performs  $O(N^2)$  FLOPs and only requires  $O(N)$  extra memory beyond inputs, output, output gradient, and input gradients.

We analyze the IO-complexity of the backward pass, similar to the forward pass (Theorem 2).

**Theorem 5.** Let  $N$  be the sequence length,  $d$  be the head dimension, and  $M$  be size of SRAM with  $d \leq M \leq Nd$ . Standard attention (Algorithm 0) backward pass requires  $\Theta(Nd + N^2)$  HBM accesses, while FLASHATTENTION backward pass (Algorithm 4) requires  $\Theta(N^2d^2M^{-1})$  HBM accesses.

The proof is in Appendix C.

## B.5 Comparison with Rabe and Staats [66]

We describe here some similarities and differences between our FLASHATTENTION algorithm and the algorithm of Rabe and Staats [66].

Conceptually, both FLASHATTENTION and Rabe and Staats [66] operate on blocks of the attention matrix using the well-established technique of tiling (or softmax scaling) [51, 60]. To reduce the memory footprint, both methods avoid storing the large attention matrix in the forward pass and recompute it in the backward pass.

The first major difference is that Rabe and Staats [66] focuses on the reducing the total memory footprint (maximum amount of GPU memory required) while FLASHATTENTION focuses on reducing memory accesses (the number of memory reads/writes). As mentioned in Section 2, the amount of memory access is the primary determining factor of runtime. Reducing memory accesses also necessarily reduces the total amount of memory required (e.g., if an operation incurs  $A$  memory accesses, then its total memory requirement is at most  $A$ ). As a result, FLASHATTENTION is faster than standard attention ( $2\text{-}4\times$ ) while Rabe and Staats [66] is around the same speed or slightly slower than standard attention. In terms of total memory required, both methods offer substantial memory saving.

The second difference between the two methods is the way information is summarized from each block to pass to the next block. Rabe and Staats [66] summarizes each block with its temporary output along with the softmax normalization statistics. At the end of the forward pass, the temporary outputs of all the blocks are combined using the statistics to produce the final output. FLASHATTENTION instead incrementally updates the output (Algorithm 1 line 12) after processing each block, so only one copy of the output is needed (instead of  $K$  copies for  $K$  blocks). This means that FLASHATTENTION has smaller total memory requirement compared to Rabe and Staats [66].

The final major difference is the way the backward pass is computed. Rabe and Staats [66] uses gradient checkpointing to recompute the attention matrix and the temporary output of each block. FLASHATTENTION instead simplifies the backward pass analytically (Appendices B.2 and B.4). It only recomputes the attention matrix and does not recompute the temporary output of each block. This reduces the memory requirement for the backward pass and yields speedup.

## C Proofs

*Proof of Theorem 1.* We first count the number of FLOPs and extra memory required.

The dominating FLOPs are from matrix multiplication. In the inner loop, (Algorithm 1 line 9), we compute  $\mathbf{Q}_i \mathbf{K}_j^\top \in \mathbb{R}^{B_r \times B_c}$  for  $\mathbf{Q}_i \in \mathbb{R}^{B_r \times d}$  and  $\mathbf{K}_j \in \mathbb{R}^{B_c \times d}$ , which takes  $O(B_r B_c d)$  FLOPs. We also compute (Algorithm 1 line 12)  $\tilde{\mathbf{P}}_{ij} \mathbf{V}_j \in \mathbb{R}^{B_r \times d}$  for  $\tilde{\mathbf{P}}_{ij} \in \mathbb{R}^{B_r \times B_c}$  and  $\mathbf{V}_j \in \mathbb{R}^{B_c \times d}$ , which takes  $O(B_r B_c d)$  FLOPs. We execute the inner loops  $T_c T_r = \left\lceil \frac{N}{B_c} \right\rceil \left\lceil \frac{N}{B_r} \right\rceil$  times. Therefore the total number of FLOPs is

$$O\left(\frac{N^2}{B_c B_r} B_r B_c d\right) = O(N^2 d).$$

In terms of extra memory required, we see that we need  $O(N)$  memory to store the statistics  $(\ell, m)$ .

We now prove the algorithm's correctness by induction on  $j$  for  $0 \leq j \leq T_c$ . Let  $\mathbf{K}_{:,j} \in \mathbb{R}^{j B_c \times d}$  be the first  $j B_c$  rows of  $\mathbf{K}$ , and similarly  $\mathbf{V}_{:,j} \in \mathbb{R}^{j B_c \times d}$  the the first  $j B_c$  rows of  $\mathbf{V}$ . Let  $\mathbf{S}_{:,j} = \mathbf{Q} \mathbf{K}_{:,j}^\top \in \mathbb{R}^{N \times j B_c}$ , and  $\mathbf{P}_{:,j} = \text{softmax}(\mathbf{S}_{:,j}) \in \mathbb{R}^{N \times j B_c}$  (softmax applied row-wise). Let  $m^{(j)}, \ell^{(j)}, \mathbf{O}^{(j)}$  be the values of  $m, \ell, \mathbf{O}$  in HBM after the  $j$ -th iteration of the outer loop (Algorithm 1 line 5). (Note that these values of  $m, \ell, \mathbf{O}$  are updated after each iteration of the outer loop.) We want to show that after the  $j$ -th iteration of the outer loop, we have computed in HBM:

$$m^{(j)} = \text{rowmax}(\mathbf{S}_{:,j}) \in \mathbb{R}^N, \quad \ell^{(j)} = \text{rowsum}(\exp(\mathbf{S}_{:,j} - m^{(j)})) \in \mathbb{R}^N, \quad \mathbf{O}^{(j)} = \mathbf{P}_{:,j} \mathbf{V}_{:,j} \in \mathbb{R}^{N \times d}.$$

Based on our initialization (Algorithm 1 line 2), this claim is true for  $j = 0$  (i.e., before the any iteration of the outer loop is executed). Suppose that the claim holds for some  $j = 0, \dots, T_c - 1$ . We want to show that the claim also holds for  $j + 1$ . Indeed, when we update the statistics in the inner loop (Algorithm 1 line 10)

on the  $(j+1)$ -th iteration of the outer loop, we update  $m^{(j+1)} = \max(m^{(j)}, \tilde{m})$  where  $\tilde{m} \in \mathbb{R}^N$  is the row-max of  $\mathbf{S}_{:,j:j+1}$ , the slice of  $\mathbf{S}$  from column  $jB_c$  to column  $(j+1)B_c - 1$ . This implies that

$$m^{(j+1)} = \text{rowmax}(\mathbf{S}_{:,j:j+1}) \in \mathbb{R}^N.$$

Similarly, we update

$$\ell^{(j+1)} = e^{m^{(j)} - m^{(j+1)}} \ell^{(j)} + e^{\tilde{m} - m^{(j+1)}} \tilde{\ell},$$

where  $\tilde{\ell} = \text{rowsum}(\exp(\mathbf{S}_{:,j:j+1} - \tilde{m})) \in \mathbb{R}^N$ . By the same algebraic manipulation in Section 3.1, we obtain:

$$\ell^{(j+1)} = \text{rowsum}(\exp(\mathbf{S}_{:,j:j+1} - m^{(j+1)})) \in \mathbb{R}^N.$$

Let  $\mathbf{V}_{j:j+1}$  be the slice of  $\mathbf{V}$  from column  $jB_c$  to column  $(j+1)B_c - 1$ , we also update:

$$\begin{aligned} \mathbf{O}^{(j+1)} &= \text{diag}(\ell^{(j+1)})^{-1} (\text{diag}(\ell^{(j)}) e^{m^{(j)} - m^{(j+1)}} \mathbf{O}^{(j)} + e^{\tilde{m} - m^{(j+1)}} \exp(\mathbf{S}_{j:j+1} - \tilde{m}) \mathbf{V}_{j:j+1}) \\ &= \text{diag}(\ell^{(j+1)})^{-1} (\text{diag}(\ell^{(j)}) e^{m^{(j)} - m^{(j+1)}} \mathbf{P}_{:,j} \mathbf{V}_{:,j} + e^{-m^{(j+1)}} \exp(\mathbf{S}_{j:j+1}) \mathbf{V}_{j:j+1}) \\ &= \text{diag}(\ell^{(j+1)})^{-1} (\text{diag}(\ell^{(j)}) e^{m^{(j)} - m^{(j+1)}} \text{diag}(\ell^{(j)}) \exp(\mathbf{S}_{:,j} - m^{(j)}) \mathbf{V}_{:,j} + e^{-m^{(j+1)}} \exp(\mathbf{S}_{j:j+1}) \mathbf{V}_{j:j+1}) \\ &= \text{diag}(\ell^{(j+1)})^{-1} (e^{-m^{(j+1)}} \exp(\mathbf{S}_{:,j}) \mathbf{V}_{:,j} + e^{-m^{(j+1)}} \exp(\mathbf{S}_{j:j+1}) \mathbf{V}_{j:j+1}) \\ &= \text{diag}(\ell^{(j+1)})^{-1} (\exp(\mathbf{S}_{:,j} - m^{(j+1)}) \mathbf{V}_{:,j} + \exp(\mathbf{S}_{j:j+1} - m^{(j+1)}) \mathbf{V}_{j:j+1}) \\ &= \text{diag}(\ell^{(j+1)})^{-1} \left( \exp \left( [\mathbf{S}_{:,j} \quad \mathbf{S}_{j:j+1}] - m^{(j+1)} \right) \right) \begin{bmatrix} \mathbf{V}_{:,j} \\ \mathbf{V}_{j:j+1} \end{bmatrix} \\ &= \text{softmax}(\mathbf{S}_{:,j+1}) \mathbf{V}_{j:j+1}. \end{aligned}$$

We then see that the claim is also true for  $j+1$ . By induction, the claim is true for all  $j = 0, \dots, T_c$ .

When  $j = T_c$ , we conclude that the final value of  $\mathbf{O}$  in HBM is  $\text{softmax}(\mathbf{S})\mathbf{V} = \text{softmax}(\mathbf{Q}\mathbf{K}^\top)\mathbf{V}$ .

□

*Proof of Theorem 2.* We first analyze the IO complexity of standard attention implementation. The inputs  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$  reside in HBM, and at the end of the algorithm the output  $\mathbf{O} \in \mathbb{R}^{N \times d}$  is written to HBM.

In the first step of computing the matrix multiply  $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$ , the inputs  $\mathbf{Q}, \mathbf{K}$  are read from HBM and the output  $\mathbf{S} \in \mathbb{R}^{N \times N}$  is written to HBM (Algorithm 0 line 1). This incurs  $\Theta(Nd + N^2)$  HBM accesses.

In the second step of computing  $\mathbf{P} = \text{softmax}(\mathbf{S})$ , the input  $\mathbf{S}$  is read from HBM and the output  $\mathbf{P}$  is written to HBM (Algorithm 0 line 2). This incurs  $\Theta(N^2)$  HBM accesses.

In the last step of computing  $\mathbf{O} = \mathbf{P}\mathbf{V}$ , the inputs  $\mathbf{P}, \mathbf{V}$  are read from global memory and the output  $\mathbf{O}$  is written to HBM (Algorithm 0 line 3). This incurs  $\Theta(Nd + N^2)$  HBM accesses.

Overall, standard attention implementation requires  $\Theta(Nd + N^2)$  global memory accesses.

We now analyze the IO complexity of streaming attention.

Following Algorithm 1, we see that each element of  $\mathbf{K}$  and  $\mathbf{V}$  is loaded from HBM once (Algorithm 1 line 6). We make  $T_c$  passes over  $\mathbf{Q}$  and  $\mathbf{O}$ , each pass loading all of  $\mathbf{Q}$  and all of  $\mathbf{O}$  to HBM (Algorithm 1 line 8). Therefore the number of HBM accesses is  $\Theta(Nd + NdT_c) = \Theta(NdT_c)$ .

We derive the conditions on the block sizes  $B_c$  and  $B_r$ . We need the blocks  $\mathbf{K}_j$  and  $\mathbf{V}_j$  of size  $B_c \times d$  to fit into on-chip memory, which translates to:

$$B_c d = O(M) \Leftrightarrow B_c = O\left(\frac{M}{d}\right).$$

Similarly, we need the blocks  $\mathbf{Q}_i, \mathbf{O}_i$  of size  $B_r \times d$  to fit into on-chip memory, which translates to:

$$B_r d = O(M) \Leftrightarrow B_r = O\left(\frac{M}{d}\right).$$

Finally, we need the block  $\mathbf{S}_{ij}$  of size  $B_r \times B_c$  to fit into on-chip memory, which translates to:

$$B_r B_c = O(M).$$

We therefore set:

$$B_c = \Theta\left(\frac{M}{d}\right), \quad B_r = \Theta\left(\min\left(\frac{M}{d}, \frac{M}{B_c}\right)\right) = \Theta\left(\min\left(\frac{M}{d}, d\right)\right).$$

We then have:

$$T_c = \frac{N}{B_c} = \Theta\left(\frac{Nd}{M}\right).$$

As a result, the number of HBM accesses is:

$$\Theta(NdT_c) = \Theta\left(\frac{N^2d^2}{M}\right).$$

□

*Proof of Proposition 3.* For contradiction, suppose that there exists an algorithm that computes exact attention where the number for HBM access for all  $M \in [d, Nd]$  is

$$o\left(\frac{N^2d^2}{M}\right).$$

In the regime of  $M = \Theta(Nd)$ , this results in the number of HBM accesses:

$$o\left(\frac{N^2d^2}{Nd}\right) = o(Nd).$$

However, the input to attention (matrices  $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ ) and the output  $\mathbf{O}$  have size  $Nd$  and they start out being in HBM, so if the algorithm computes exact attention it must incur at least  $\Omega(Nd)$  HBM accesses. This is a contradiction. □

*Proof of Theorem 5.* The IO complexity of the attention backward is very similar to the IO complexity of the attention forward (Theorem 2). Here we provide a sketch of the proof.

We first analyze the IO complexity of standard attention backward pass. The inputs  $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{dO} \in \mathbb{R}^{N \times d}$  reside in HBM, and the at the end of the algorithm the outputs  $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV} \in \mathbb{R}^{N \times d}$  are written to HBM.

At each step of the standard attention backward pass, one needs to load inputs of size  $Nd$  or  $N^2$  from HBM, and needs to write the outputs of size  $N^2$  or  $Nd$  to HBM. This incurs  $\Theta(Nd + N^2)$  HBM accesses.

We now analyze the IO complexity of FLASHATTENTION backward pass.

Similar to Theorem 2, we see that each element of  $\mathbf{K}$  and  $\mathbf{V}$  is loaded from HBM once. Each element of  $\mathbf{dK}$  and  $\mathbf{dV}$  is only written to HBM once. We make  $T_c$  passes over  $\mathbf{Q}, \mathbf{O}, \mathbf{dO}$ , each pass loading all of  $\mathbf{Q}, \mathbf{O}, \mathbf{dO}$  to HBM. We also make  $T_c$  passes over  $\mathbf{dQ}$ , each pass reading/writing all of  $\mathbf{dQ}$  from/to HBM. Therefore the number of HBM accesses is  $\Theta(Nd + NdT_c) = \Theta(NdT_c)$ .

As in the proof of Theorem 2, the constraints on the block sizes are that:

$$B_c = \Theta\left(\frac{M}{d}\right), \quad B_r = \Theta\left(\min\left(\frac{M}{d}, d\right)\right).$$

We then have:

$$T_c = \frac{N}{B_c} = \Theta\left(\frac{Nd}{M}\right).$$

As a result, the number of HBM accesses is:

$$\Theta(NdT_c) = \Theta\left(\frac{N^2d^2}{M}\right).$$

□

---

**Algorithm 5** Block-Sparse FLASHATTENTION Forward Pass

---

**Require:** Matrices  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$  in HBM, on-chip SRAM of size  $M$ , softmax scaling constant  $\tau \in \mathbb{R}$ , masking function MASK, dropout probability  $p_{\text{drop}}$ , block sizes  $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$ , block sparsity mask  $M \in \{0, 1\}^{N/B_r \times N/B_c}$ .

- 1: Initialize the pseudo-random number generator state  $\mathcal{R}$  and save to HBM.
- 2: Initialize  $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$  in HBM.
- 3: Divide  $\mathbf{Q}$  into  $T_r = \lceil \frac{N}{B_r} \rceil$  blocks  $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$  of size  $B_r \times d$  each, and divide  $\mathbf{K}, \mathbf{V}$  in to  $T_c = \lceil \frac{N}{B_c} \rceil$  blocks  $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$  and  $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$ , of size  $B_c \times d$  each.
- 4: Divide  $\mathbf{O}$  into  $T_r$  blocks  $\mathbf{O}_1, \dots, \mathbf{O}_{T_r}$  of size  $B_r \times d$  each, divide  $\ell$  into  $T_r$  blocks  $\ell_1, \dots, \ell_{T_r}$  of size  $B_r$  each, divide  $m$  into  $T_r$  blocks  $m_1, \dots, m_{T_r}$  of size  $B_r$  each.
- 5: **for**  $1 \leq j \leq T_c$  **do**
- 6:   Load  $\mathbf{K}_j, \mathbf{V}_j$  from HBM to on-chip SRAM.
- 7:   **for**  $1 \leq i \leq T_r$  **do**
- 8:     **if**  $M_{ij} \neq 0$  **then**
- 9:       Load  $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$  from HBM to on-chip SRAM.
- 10:       On chip, compute  $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$ .
- 11:       On chip, compute  $\mathbf{S}_{ij}^{\text{masked}} = \text{MASK}(\mathbf{S}_{ij})$ .
- 12:       On chip, compute  $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}^{\text{masked}}) \in \mathbb{R}^{B_r}, \tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij}^{\text{masked}} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$  (pointwise),  $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$ .
- 13:       On chip, compute  $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}, \ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$ .
- 14:       On chip, compute  $\tilde{\mathbf{P}}_{ij}^{\text{dropped}} = \text{dropout}(\tilde{\mathbf{P}}_{ij}, p_{\text{drop}})$ .
- 15:       Write  $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij}^{\text{dropped}} \mathbf{V}_j)$  to HBM.
- 16:       Write  $\ell_i \leftarrow \ell_i^{\text{new}}, m_i \leftarrow m_i^{\text{new}}$  to HBM.
- 17:     **end if**
- 18:   **end for**
- 19: **end for**
- 20: Return  $\mathbf{O}, \ell, m, \mathcal{R}$ .

---

## D Extension Details

### D.1 Block-sparse FLASHATTENTION

We describe the full block-sparse FLASHATTENTION algorithm in Algorithm 5. The algorithm is identical to Algorithm 2, except that we skip zero blocks.

We prove the IO-complexity of block-sparse FLASHATTENTION.

*Proof of Proposition 4.* The proof is very similar to the proof of Theorem 2. For the block-sparse case, notice that we only need to load blocks corresponding to nonzero blocks. As a result, the number of HBM accesses are scaled by  $s$ , the fraction of nonzero blocks in the block-sparsity mask. However, for small values of  $s$ , we would still need to write the result  $\mathbf{O} \in \mathbb{R}^{N \times d}$ . Therefore the number of HBM accesses is

$$\Theta\left(Nd + \frac{N^2d^2}{M}s\right).$$

□

### D.2 Potential Extensions

We discuss here a few potential extensions of the IO-aware approach to speed up deep learning training.

**Multi-GPU Attention.** Large language models are trained on hundreds or thousands of GPUs, and one typically splits the attention computation between 4-8 GPUs on the same node [77]. This introduces another level of memory hierarchy: beside GPU SRAM and GPU HBM, we also have the HBM of other

GPUs. For very long sequences, the different GPUs on the same node can cooperate to compute attention by taking into account the asymmetry of different levels of memory hierarchy.

**Sparse MLP layers.** Typical dense MLP layers are compute-bound and not memory-bound. To improve their efficiency, MLP layers with sparse weight matrices can be used [17]. However, many sparse MLP layers are instead memory-bound, and their speedup is often not proportional to the sparsity. We believe that an IO-aware implementation can alleviate this issue and realize the benefits of sparsity. We are excited about future work in this direction, to reduce the computational requirement of large models and improve their wall-block runtime.

**Kernel machine learning.** Our approach in FLASHATTENTION relies on the fact that the  $N \times N$  attention matrix is a function of a low-rank matrix  $\mathbf{Q}\mathbf{K}^\top$  (of rank  $d \ll N$ ). As a result, we can repeatedly load the inputs  $\mathbf{Q}, \mathbf{K}$  and recompute the block of the attention matrix that we need, significantly reducing HBM access. A similar scenario happens in kernel machine learning: each element  $K_{ij}$  of the  $N \times N$  kernel matrix  $\mathbf{K}$  is a function of two vectors of size  $d \ll N$ , as it measures the similarity between two datapoints  $x_i$  and  $x_j$ . The KeOps library [8, 26] is a successful example of how reducing memory reads/writes can speed up kernel operations. We hope that this will motivate kernel methods that focus more on reducing IOs instead of just FLOPs.

## E Full Experimental Results

### E.1 BERT

We train BERT-large following the training procedure and hyperparameters of the reference MLPerf 1.1 implementation. In particular, we use the LAMB optimizer with learning rate 3.75e-3, with batch size 448, trained for at most 7100 steps. The training is stopped once the validation accuracy (for masked language modeling) reaches the target 72.0%, and the wall-clock run-time is measured. We train with FP16 precision using Apex AMP (with O2 optimization level).

We compare our results with the reported training speed from Nvidia that was submitted to MLPerf 1.1 (Table 1).

We use the same train / validation data split provided by MLPerf 1.1 reference implementation. In particular, we evaluate on the same 10000 validation examples as the baseline from Nvidia.

We train the model on 8xA100-80GB GPUs. Each training run takes between 16 and 19 minutes, and we average the results of 10 runs.

### E.2 GPT-2

We use the standard implementations of GPT-2 [67] from Huggingface `transformers` library and from Nvidia’s Megatron-LM repo. We follow the training recipe of the Megatron-LM repo.

We use an effective batch size of 512, and use gradient accumulation to fit into available GPU memory. We use the AdamW optimizer, with learning rate 6e-4 for GPT-2 small and 1.5e-4 for GPT-2 medium, and weight decay of 0.1. All models are trained with the same hyperparameters for 400K steps. We run all implementations with mixed-precision training (PyTorch AMP).

We use the Openwebtext dataset, with the GPT-2 BPE tokenizer. We randomly select 0.5% of the dataset as the validation set, with the rest being used as training set. This random selection of validation set is done once, and all models are evaluated on the same validation set.

We train the model on 8xA100-40GB GPUs, and we measure the wall-clock training time. Training GPT-2 small takes between 2.7-9.5 days, and training GPT-2 medium takes between 6.9-21.0 days (Table 2).

In Fig. 4, we plot of the validation perplexity throughout training of GPT-2 small/medium, using either HuggingFace implementation or our FLASHATTENTION implementation. We see that FLASHATTENTION behaves the same as the baseline implementation and the validation perplexity curves of the two implementations almost lie on top of each other.

**Long Document Classification.** For MIMIC-III and ECtHR, we follow the hyperparameters of Dai et al. [13].

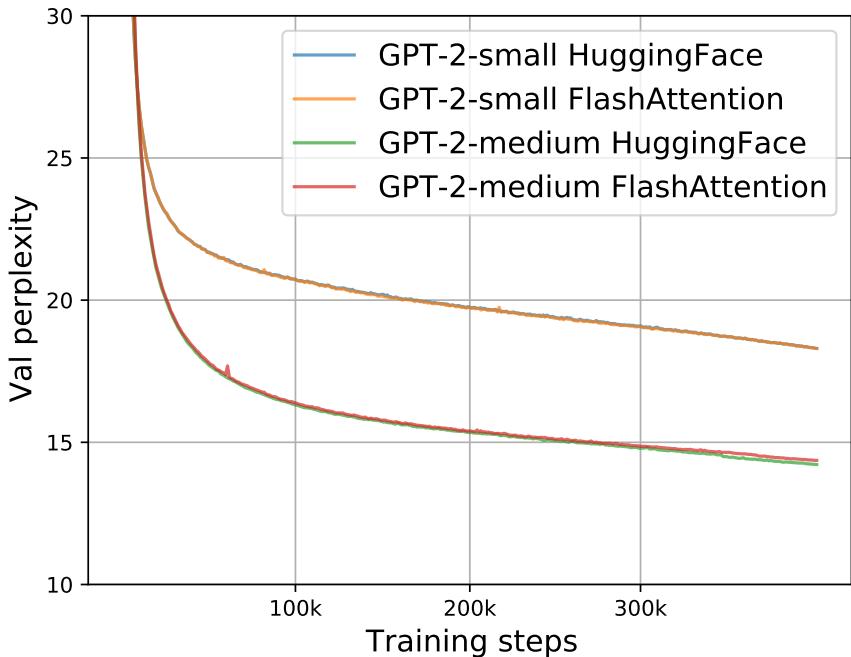


Figure 4: Validation perplexity of GPT-2 small/medium using two implementations. We confirm that FLASHATTENTION yields the same validation curves as the baseline implementation from HuggingFace.

### E.3 LRA details

We follow the hyperparameters from the Long-range arena paper [80], the Long-range arena repo (<https://github.com/google-research/long-range-arena>), and the Nyströmformer reproduction [90]. To be generous to the baseline methods, if we are unable to reproduce the performance of any baseline for any of the five tasks, we report the better performance from Tay et al. [80] or Xiong et al. [90] for that baseline on that task.

After hyperparameter tuning, almost all of the attention methods achieve similar accuracy on all of the five LRA tasks.

We run all methods with mixed-precision training, except for Performer (not stable with mixed precision) and Local Attention (implementation does not support FP16).

To calculate the overall wallclock-time speedup, we take the geometric mean of the wallclock-time speedup of each of the five tasks.

**Path-X** For Path-X and Path-256, we follow the hyperparameters from the PathFinder-32 experiments from the long-range arena paper[80]. For both, we first pretrain a model on Path-64. We take the checkpoint after 200 epochs, upsample its positional embedding (we duplicate the positional embeddings gridwise in space), and fine-tune it on the downstream task for 200 epochs with one epoch of linear warmup, and cosine decay of the learning rate. For Path-X, we take the best performing checkpoint (according to val accuracy), and additionally fine-tune it for 200 epochs with the same warmup and learning rate (this adds roughly 4 points of accuracy to FLASHATTENTION for Path-X, but the model starts overfitting afterwards).

### E.4 Comparison with Apex FMHA

We compare our method/implementation with Apex FMHA (<https://github.com/NVIDIA/apex/tree/master/apex/contrib/csrc/fmha>).

When we started this project, Apex FMHA was the fastest implementation of attention (that we knew of), tailored for short sequences of length at most 512. In fact, almost all MLPerf submissions for BERT training benchmark running on Nvidia GPUs use FMHA for their model code, as of MLPerf 1.1 [58]. Since

Table 7: Runtime (ms) of FLASHATTENTION compared to FMHA by sequence length, with masking and dropout, measured on an A100-SXM4-40GB GPU. Batch size 64, 16 heads, head dimension 64 (i.e., BERT-large size).

Attention Method	128	256	512
Apex FMHA forward	0.10	0.29	1.14
FLASHATTENTION forward	<b>0.08</b>	<b>0.22</b>	<b>0.81</b>
Apex FMHA backward	<b>0.17</b>	<b>0.52</b>	<b>1.81</b>
FLASHATTENTION backward	0.20	0.53	2.00
Apex FMHA forward + backward	<b>0.27</b>	0.81	2.95
FLASHATTENTION forward + backward	0.28	<b>0.75</b>	<b>2.81</b>

FMHA targets BERT models, it only supports head dimension 64, and only runs on A100 GPUs. FMHA fuses the attention computation dropout( $\text{softmax}(\text{MASK}(\mathbf{Q}\mathbf{K}^T))\mathbf{V}$ ) into one CUDA kernel. In the forward pass, it stores the attention matrix  $\text{softmax}(\text{MASK}(\mathbf{Q}\mathbf{K}^T))$  to HBM to be used in gradient computation. As a result, it does not offer substantial memory saving (though for shorter sequences memory footprint is often not a primary concern).

We use FMHA code as a starting point, and apply two well-established techniques (tiling and recomputation) to deal with long sequences and to save memory as mentioned in Section 3. As a result, we can support much longer sequences (e.g., up to length 64K). We also support more head dimensions (16, 32, 64, 128) and broader GPU types (all Turing and Ampere GPUs at the time of writing).

In Table 7, we compare the performance of FLASHATTENTION and Apex FMHA for short sequences (as FMHA only supports sequence length at most 512). Generally FLASHATTENTION is slightly faster than FMHA in the forward pass and slightly slower than FMHA in the backward pass. This is because we do not store the attention matrix in the forward pass and recompute it in the backward pass. Compared to FMHA, the overall runtime of FLASHATTENTION is about 4% slower for sequence length 128, 8% faster for sequence length 256, and 5% faster for sequence length 512.

## E.5 Speedup On Different Hardware and Configurations

Speedup varies between different types of GPU types and generations depending on HBM bandwidth and SRAM size. In this section, we profile FLASHATTENTION speedup on different GPUs and configurations.

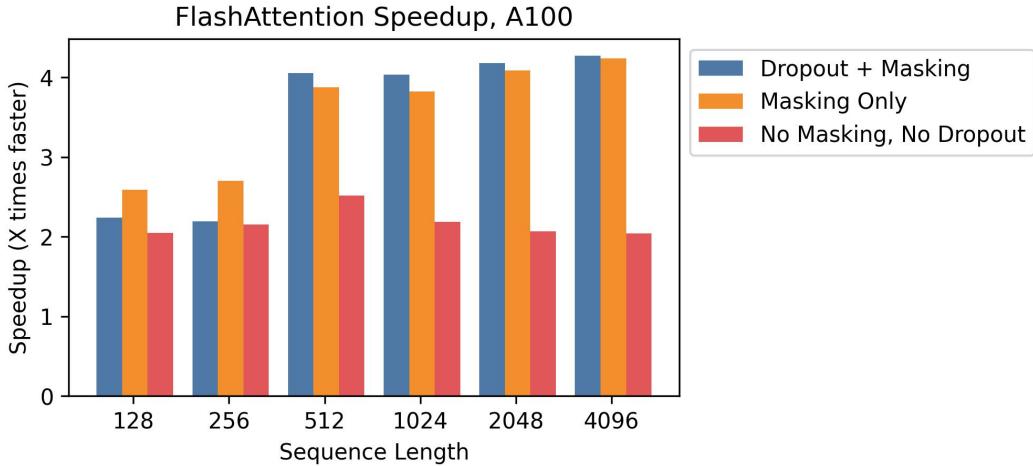


Figure 5: Speedup over standard PyTorch attention at different sequence lengths, on A100.

**A100** Figure 5 shows speedup on an A100 GPU with batch size 8, head dimension 64, and 12 attention heads, across different sequence lengths. We generally see 2-4× speedup, and we see more speedup when using dropout and masking due to kernel fusion.

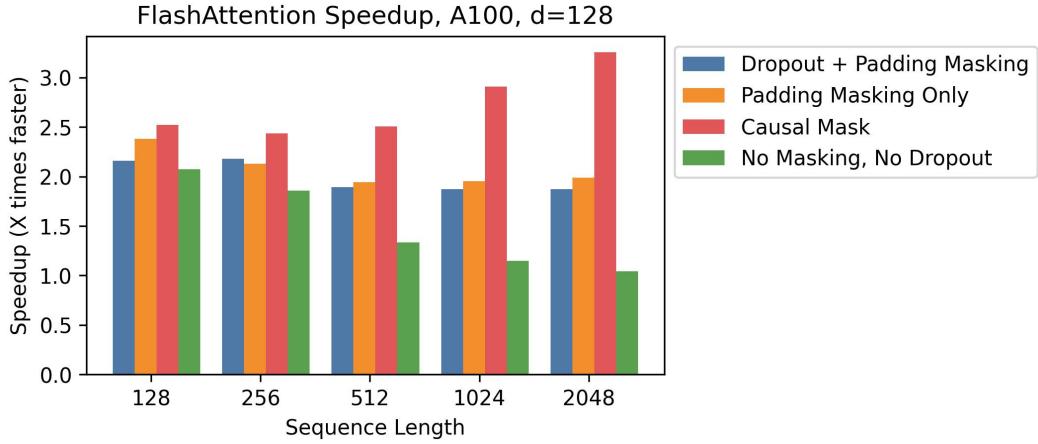


Figure 6: Speedup over standard PyTorch attention at different sequence lengths, on A100, with head dimension 128.

**A100, Head Dimension 128** Speedup also changes when we increase the head dimension. Each block requires more memory, so we need to use smaller block sizes to fit into SRAM. Figure 6 shows speedup with head dimension 128 on an A100 (batch size 16, 12 heads). We see less speedup overall—but we can still see significant speedup (up to 3×) with a causal mask, where half the blocks are masked out.



Figure 7: Speedup over standard PyTorch attention at different sequence lengths, on RTX 3090.

**RTX 3090** Figure 7 shows speedup on an RTX 3090 GPU. Here, we use batch size 12 with 12 attention heads. We observe slightly higher speedups on the RTX 3090 (between 2.5–4.5×), since the memory bandwidth on an RTX 3090 is lower than on an A100 (roughly 900 GB/s vs. 1.5 TB/s).

**T4** Figure 8 shows speedup on a T4 GPU. T4 SRAM is smaller than A100, so we need to make the block sizes smaller in FLASHATTENTION. As a result, we observe less speedup on T4, which matches the IO complexity analysis in Section 3.2. T4 GPUs are commonly used for inference, so we also report speedup on the forward pass only.

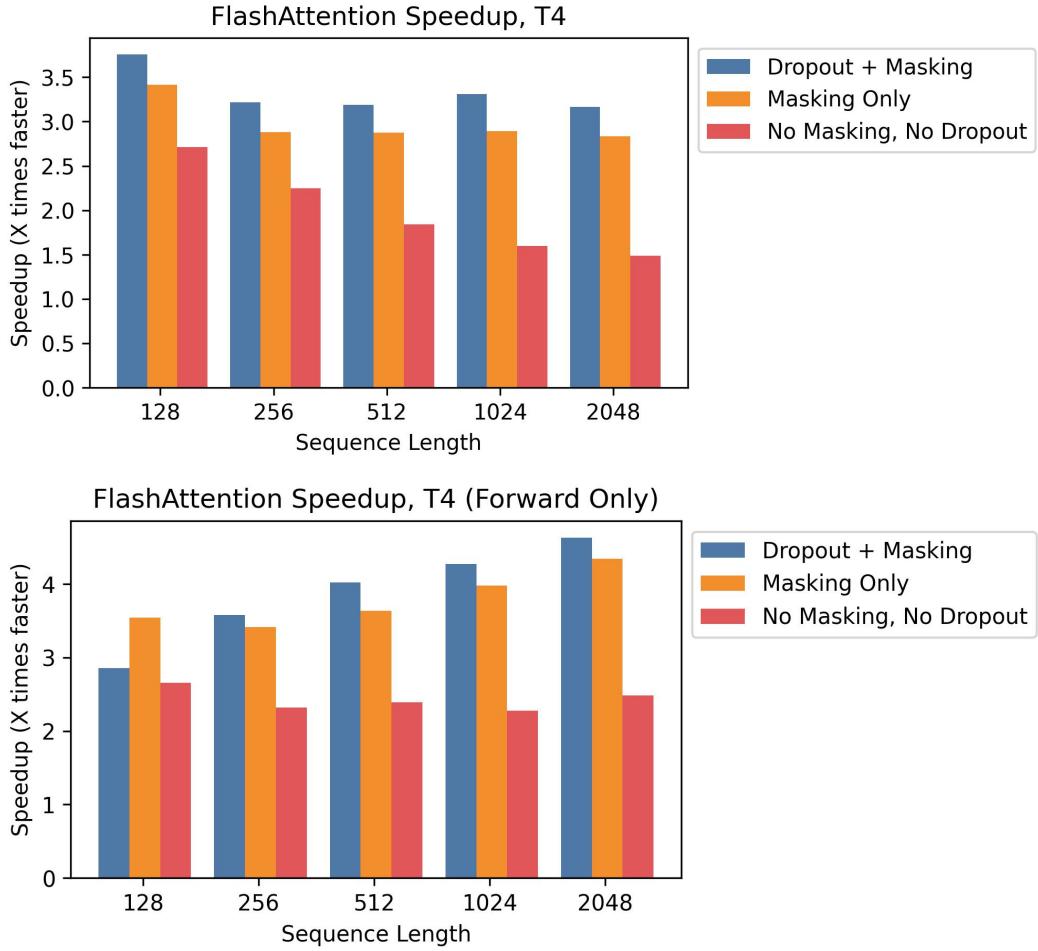


Figure 8: Speedup over standard PyTorch attention at different sequence lengths, on T4. **Top:** Combined forward pass + backward pass. **Bottom:** Forward pass only.

## E.6 Full Benchmarking Results

We report the full benchmarking results and experimental details on A100.

**Baselines** We compare against reference implementations for exact attention from PyTorch/HuggingFace and Megatron, approximate attention, and sparse attention. For approximate attention, we compare against reference implementations of Reformer [51], Local Attention [68], Linformer Attention [84], Smyrf [19], and LongShortFormer (LSFormer) [94]. For sparse attention, we compare against reference implementations of Block-Sparse Attention from OpenAI [11], Longformer[3], and BigBird Attention [92]. For the approximate and sparse attention, we use a compression ratio of 1/8, or a compressed sequence length of 256, whichever is smaller.

**Setup** We measure runtime and memory usage of the attention computation with 8 heads of dimension 64, and batch size 16 on a machine with one A100 GPU with 40 GB of GPU HBM. We vary sequence length in our experiments. We compute attention on random vectors for  $\mathbf{Q}$ ,  $\mathbf{K}$ , and  $\mathbf{V}$  (we do not measure the projection from the hidden layer). For dropout, we use dropout 0.1; for masking, we use a padding mask with uniformly-random mask lengths between the total sequence length and the total sequence length minus 20. To measure runtime, we take the average of 100 measurements of the attention call. We only measure memory footprint once, since it does not vary between runs.

Table 8: Pointers to results tables.

Dropout	Masking	Pass	Table
Yes	Yes	Forward	Table 9
Yes	Yes	Backward	Table 10
Yes	Yes	Combined	Table 11
No	Yes	Forward	Table 12
No	Yes	Backward	Table 13
No	Yes	Combined	Table 14
Yes	No	Forward	Table 15
Yes	No	Backward	Table 16
Yes	No	Combined	Table 17
No	No	Forward	Table 18
No	No	Backward	Table 19
No	No	Combined	Table 20
No	No	Memory Usage (Combined)	Table 21

 Table 9: Forward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length, **with dropout and masking**. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
<b>PyTorch Attention</b>	0.36	0.34	0.78	2.54	9.33	36.33	-	-	-	-
	0.40	0.40	1.10	3.65	16.19	-	-	-	-	-
<b>Local Attention</b>	2.03	3.15	5.67	11.02	22.59	46.14	97.38	212.13	-	-
	0.83	0.86	1.01	2.20	7.13	14.32	28.60	57.79	117.67	-
	0.67	0.52	0.69	<u>0.71</u>	<u>1.65</u>	<u>3.18</u>	<u>6.15</u>	<u>12.16</u>	<u>24.17</u>	<u>52.39</u>
	2.27	2.34	3.91	7.44	14.71	29.22	58.27	116.41	-	-
<b>Reformer</b>	1.18	1.27	1.34	3.38	11.40	22.55	44.95	89.76	179.66	-
<b>Linformer</b>	-	-	-	-	-	-	-	-	-	-
<b>Smyrf</b>	-	-	-	-	-	-	-	-	-	-
<b>LSformer</b>	-	-	-	-	-	-	-	-	-	-
<b>Block Sparse</b>	1.12	1.11	2.13	2.77	6.95	20.91	-	-	-	-
<b>Longformer</b>	1.22	1.14	1.08	1.95	5.72	12.98	-	-	-	-
<b>BigBird</b>	1.13	1.12	1.12	1.77	6.03	13.68	-	-	-	-
<b>FLASHATTENTION</b>	<b>0.04</b>	<b>0.06</b>	<b>0.21</b>	<b>0.82</b>	<b>2.85</b>	<b>10.41</b>	<b>41.74</b>	<b>167.19</b>	<b>670.76</b>	<b>2682.35</b>
<b>Block-Sparse FLASHATTENTION</b>	<u>0.06</u>	<u>0.06</u>	<u>0.06</u>	<u>0.12</u>	<u>0.44</u>	<u>0.86</u>	<u>1.70</u>	<u>3.29</u>	<u>6.55</u>	<u>13.34</u>

We report timing results on the forward pass, backward pass, and combined forward + backward pass. We measure each method with and without dropout, masking, or both—except for Block Sparse, Longformer, and BigBird. These methods did not successfully run the backward pass with masking due to a bug in external libraries, so we measured them without masking to be generous. We use FP16 for all measurements, except for Local Attention, whose implementation only supports FP32.

For each baseline, we increase sequence length until it runs out of memory on the GPU, except for the following exceptions: The Megatron implementation does not support sequence lengths longer than 2048. Block-Sparse (OpenAI) does not support sequence lengths longer than 4096. Longformer and BigBird do not support sequence lengths longer than 8092.

We measure memory usage on the combined forward + backward pass, without dropout or masking.

**Results** Table 8 summarizes all the experimental configurations and contains pointers to the results tables.

Table 10: Backward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length, **with dropout and masking**. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	0.37	0.49	1.66	5.81	22.32	<u>87.67</u>	-	-	-	-
Megatron	0.35	0.32	<u>0.77</u>	2.42	8.43	-	-	-	-	-
Reformer	2.37	4.59	8.91	17.68	35.13	70.05	140.01	-	-	-
Local Attention	0.55	0.62	1.49	4.03	13.78	27.61	55.20	110.27	221.40	-
Linformer	0.89	0.80	0.81	<u>0.93</u>	2.48	<u>4.75</u>	<u>9.29</u>	<u>18.27</u>	<u>36.53</u>	-
Smyrf	1.41	2.83	5.43	<u>10.72</u>	21.25	42.31	84.48	168.95	-	-
LSformer	1.75	1.76	3.01	7.50	20.07	39.08	76.39	150.82	-	-
Block Sparse	1.29	1.28	2.18	3.04	7.27	21.16	-	-	-	-
Longformer	1.27	1.31	1.29	2.04	5.24	10.74	25.95	-	-	-
BigBird	1.33	1.28	1.32	1.81	5.55	11.44	27.45	-	-	-
<b>FLASHATTENTION</b>	<b>0.30</b>	<b>0.26</b>	<u>0.68</u>	2.02	6.84	26.89	105.70	418.96	1666.89	6660.44
<b>Block-Sparse FLASHATTENTION</b>	<b>0.30</b>	<u>0.27</u>	<b>0.29</b>	<u>0.59</u>	<b>1.50</b>	<b>2.94</b>	<b>5.82</b>	<u>11.85</u>	<b>23.98</b>	<b>47.61</b>

Table 11: Forward pass + backward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length, **with dropout and masking**. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	0.84	0.86	2.35	8.29	31.75	124.19	-	-	-	-
Megatron	0.87	0.89	1.33	4.21	16.50	-	-	-	-	-
Reformer	4.30	7.76	14.60	28.74	57.79	116.34	237.57	-	-	-
Local Attention	1.40	1.60	2.06	6.06	20.94	42.01	84.08	168.48	339.45	-
Linformer	1.57	1.49	1.55	<u>1.60</u>	<u>4.19</u>	<u>8.04</u>	<u>15.71</u>	<u>30.92</u>	<u>61.47</u>	-
Smyrf	3.41	5.08	9.35	18.18	36.03	71.68	143.04	285.87	-	-
LSformer	3.08	3.10	4.26	10.90	31.59	61.72	121.51	241.18	-	-
Block Sparse	2.54	2.52	3.71	5.44	13.29	39.19	-	-	-	-
Longformer	2.47	2.49	2.51	3.10	10.39	22.49	60.44	-	-	-
BigBird	2.51	2.49	2.52	3.40	10.97	23.89	63.28	-	-	-
<b>FLASHATTENTION</b>	<b>0.43</b>	<b>0.41</b>	<u>0.95</u>	2.55	9.56	37.49	147.75	586.61	2339.11	9341.30
<b>Block-Sparse FLASHATTENTION</b>	<b>0.44</b>	<u>0.44</u>	<b>0.45</b>	<b>0.89</b>	<b>1.95</b>	<b>4.12</b>	<b>7.64</b>	<b>16.60</b>	<b>32.73</b>	<b>64.11</b>

Table 12: Forward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length, **with masking**. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	0.30	0.30	0.63	1.93	7.08	27.45	112.90	-	-	-
Megatron	0.45	0.41	0.43	1.52	5.80	-	-	-	-	-
Reformer	1.87	3.00	5.37	10.43	21.40	43.83	92.80	203.24	-	-
Local Attention	0.70	0.81	1.02	2.09	6.64	13.34	26.77	54.02	110.11	-
Linformer	0.63	0.50	0.67	<u>0.65</u>	<u>1.36</u>	<u>2.60</u>	<u>5.04</u>	<u>9.92</u>	<u>19.69</u>	<u>43.47</u>
Smyrf	2.38	2.32	3.76	7.16	14.14	28.09	55.98	111.73	-	-
LSformer	1.22	1.29	1.44	3.28	10.99	21.72	43.29	86.32	172.76	-
Block Sparse	0.96	1.04	1.66	2.16	5.41	16.15	-	-	-	-
Longformer	0.99	0.98	0.99	1.56	4.79	11.07	32.98	-	-	-
BigBird	0.96	1.02	1.02	1.48	5.05	11.59	34.16	-	-	-
<b>FLASHATTENTION</b>	<b>0.03</b>	<b>0.04</b>	<u>0.17</u>	0.68	2.28	8.40	33.55	134.14	537.50	2150.88
<b>Block-Sparse FLASHATTENTION</b>	<b>0.05</b>	<b>0.04</b>	<b>0.05</b>	<b>0.11</b>	<b>0.35</b>	<b>0.68</b>	<b>1.33</b>	<b>2.54</b>	<b>5.34</b>	<b>10.73</b>

Table 13: Backward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length, **with masking**. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	0.44	0.46	1.53	5.33	20.34	79.87	-	-	-	-
Megatron	0.29	0.31	0.65	1.95	6.49	-	-	-	-	-
Reformer	2.31	4.47	8.68	17.20	34.14	68.09	136.02	-	-	-
Local Attention	0.51	0.62	1.30	3.81	13.33	26.72	53.41	106.82	214.15	-
Linformer	0.76	0.81	0.94	<u>0.87</u>	<u>2.24</u>	<u>4.25</u>	<u>8.35</u>	<u>16.38</u>	<u>32.67</u>	<u>72.11</u>
Smyrf	1.34	2.77	5.30	10.46	20.73	41.27	82.41	164.86	-	-
LSformer	1.66	1.61	3.09	7.42	19.68	38.35	74.92	147.86	-	-
Block Sparse	1.24	1.25	2.04	2.91	6.78	19.67	-	-	-	-
Longformer	1.27	1.23	1.24	1.85	4.99	10.21	24.89	-	-	-
BigBird	1.43	1.50	1.44	1.69	5.25	10.86	26.26	-	-	-
<b>FLASHATTENTION</b>	<b>0.21</b>	<b>0.22</b>	<u>0.62</u>	1.84	5.77	22.25	86.21	338.91	1343.91	5361.09
<b>Block-Sparse FLASHATTENTION</b>	<b>0.22</b>	<u>0.22</u>	<b>0.26</b>	<b>0.57</b>	<b>1.55</b>	<b>3.13</b>	<b>5.98</b>	<b>12.21</b>	<b>23.49</b>	<b>47.85</b>

Table 14: Forward pass + backward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length, **with masking**. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	0.80	0.81	2.08	7.23	27.51	107.58	-	-	-	-
	0.81	0.83	1.09	3.36	12.39	-	-	-	-	-
Local Attention	4.16	7.46	14.06	27.68	55.66	112.15	229.37	-	-	-
	1.39	1.68	2.08	5.83	20.04	40.16	80.44	161.35	325.11	-
	1.51	1.42	1.56	<u>1.67</u>	<u>3.67</u>	<u>6.99</u>	<u>13.63</u>	<u>26.77</u>	<u>53.36</u>	<u>117.56</u>
	3.38	4.93	9.07	17.66	34.94	69.55	138.72	277.41	-	-
	3.08	3.10	4.26	10.90	31.59	61.72	121.51	241.18	-	-
Block Sparse	2.39	2.40	3.31	5.02	12.25	35.94	-	-	-	-
	2.36	2.34	2.38	2.94	9.83	21.35	58.12	-	-	-
	2.35	2.35	2.37	3.25	10.36	22.57	60.63	-	-	-
FLASHATTENTION	<b>0.32</b>	<b>0.30</b>	<u>0.83</u>	2.37	7.95	30.77	119.98	473.65	1883.43	7513.01
Block-Sparse FLASHATTENTION	<u>0.34</u>	<u>0.34</u>	<b>0.36</b>	<b>0.69</b>	<b>1.85</b>	<b>3.89</b>	<b>7.16</b>	<b>14.85</b>	<b>30.46</b>	<b>60.03</b>

Table 15: Forward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length, **with dropout**. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	0.26	0.24	0.57	1.80	6.56	25.34	-	-	-	-
	0.27	0.27	0.56	1.88	6.56	-	-	-	-	-
Local Attention	1.83	2.96	5.31	10.33	21.19	43.42	91.96	201.34	-	-
	0.51	0.60	0.78	2.01	6.23	12.52	25.07	50.50	102.18	-
	0.47	0.37	<u>0.49</u>	<b>0.52</b>	<u>1.37</u>	<u>2.65</u>	<u>5.12</u>	<u>10.13</u>	<u>20.25</u>	<u>44.16</u>
	2.12	2.01	3.15	5.97	11.83	23.36	46.48	92.72	-	-
	1.28	1.33	1.51	3.39	11.40	22.54	44.96	89.85	179.73	-
Block Sparse	1.03	1.00	1.72	2.39	5.96	17.88	-	-	-	-
	1.02	1.03	1.03	1.73	5.10	11.63	34.22	-	-	-
	0.99	1.03	1.01	1.58	5.36	12.27	35.56	-	-	-
FLASHATTENTION	<b>0.10</b>	<b>0.10</b>	<b>0.22</b>	0.83	2.81	10.38	41.63	167.01	668.74	2678.11
Block-Sparse FLASHATTENTION	0.54	0.51	0.68	<u>0.61</u>	<b>0.67</b>	<b>1.10</b>	<b>1.89</b>	<b>3.71</b>	<b>7.18</b>	<b>14.41</b>

Table 16: Backward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length, **with dropout**. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	0.44	0.35	0.90	2.94	10.77	41.67	-	-	-	-
	0.28	0.33	0.92	2.94	10.80	-	-	-	-	-
Local Attention	2.24	4.34	8.39	16.62	33.02	65.77	131.52	-	-	-
	0.51	0.58	1.41	3.71	12.96	25.98	51.94	103.72	207.78	-
	0.84	0.74	0.79	<u>0.85</u>	<u>2.28</u>	<u>4.37</u>	<u>8.66</u>	<u>17.02</u>	<u>33.78</u>	-
	1.27	2.56	4.90	9.66	19.16	38.13	76.17	152.39	-	-
	1.67	1.77	3.03	7.52	20.10	39.13	76.35	150.83	-	-
Block Sparse	1.27	1.36	2.15	3.04	7.27	21.18	-	-	-	-
	1.28	1.34	1.38	1.98	5.24	10.74	25.95	-	-	-
	1.48	1.47	1.50	1.81	5.57	11.38	27.43	-	-	-
FLASHATTENTION	<b>0.15</b>	<u>0.18</u>	<u>0.58</u>	1.86	6.50	26.21	104.27	416.10	1661.92	6643.01
Block-Sparse FLASHATTENTION	<u>0.17</u>	<b>0.17</b>	<b>0.17</b>	<b>0.40</b>	<b>1.10</b>	<b>2.04</b>	<b>4.43</b>	<b>9.33</b>	<b>18.28</b>	<b>37.31</b>

Table 17: Forward pass + backward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length, **with dropout**. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	0.66	<u>0.67</u>	1.43	4.82	17.47	67.29	-	-	-	-
	0.88	0.90	1.49	4.73	<u>17.41</u>	-	-	-	-	-
Local Attention	4.06	7.28	13.68	26.98	54.27	109.39	223.80	-	-	-
	1.09	1.40	1.99	5.61	19.23	38.62	77.30	154.63	311.12	-
	1.31	1.21	1.30	<u>1.39</u>	<u>3.73</u>	<u>7.15</u>	<u>14.05</u>	<u>27.69</u>	<u>55.00</u>	-
	3.00	4.37	8.05	15.66	31.04	61.64	123.04	245.65	-	-
	3.07	3.17	4.31	10.89	31.54	61.78	121.56	240.94	-	-
Block Sparse	2.54	2.52	3.71	5.44	13.29	39.19	-	-	-	-
	2.47	2.49	2.51	3.10	10.39	22.49	60.44	-	-	-
	2.51	2.49	2.52	3.40	10.97	23.89	63.28	-	-	-
FLASHATTENTION	<b>0.35</b>	<b>0.36</b>	<b>0.80</b>	2.52	9.16	36.70	146.13	583.45	2332.01	9323.63
Block-Sparse FLASHATTENTION	0.91	0.83	<u>0.94</u>	<b>0.92</b>	<b>1.83</b>	<b>3.50</b>	<b>7.02</b>	<b>13.56</b>	<b>26.71</b>	<b>53.92</b>

Table 18: Forward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	<u>0.21</u>	<u>0.22</u>	0.43	1.27	4.32	16.47	67.77	-	-	-
	0.24	0.26	0.42	1.33	4.28	-	-	-	-	-
Reformer	1.77	2.82	5.01	9.74	20.03	41.11	87.39	192.40	-	-
	0.48	0.57	0.80	1.90	5.76	11.56	23.13	46.65	94.74	-
Local Attention	0.46	0.36	0.45	<b>0.50</b>	<u>1.09</u>	<u>2.09</u>	<u>4.01</u>	<u>7.90</u>	<u>15.70</u>	<u>35.40</u>
	1.94	1.96	3.01	5.69	11.26	22.23	44.21	88.22	-	-
Linformer	1.21	1.34	1.34	3.31	11.01	21.71	43.27	86.32	172.85	-
	0.96	1.04	1.66	2.16	5.41	16.15	-	-	-	-
Smyrf	0.99	0.98	0.99	1.56	4.79	11.07	32.98	-	-	-
	0.96	1.02	1.02	1.48	5.05	11.59	34.16	-	-	-
<b>FLASHATTENTION</b>	<b>0.08</b>	<b>0.09</b>	<b>0.18</b>	0.68	2.40	8.42	33.54	134.03	535.95	2147.05
<b>Block-Sparse FLASHATTENTION</b>	0.56	0.52	0.63	<u>0.65</u>	<b>0.61</b>	<b>0.96</b>	<b>1.69</b>	<b>3.02</b>	<b>5.69</b>	<b>11.77</b>

Table 19: Backward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	0.26	0.29	0.78	2.44	8.82	33.87	-	-	-	-
	0.29	0.30	0.80	2.59	8.86	-	-	-	-	-
Reformer	2.18	4.21	8.14	16.12	32.02	63.84	127.60	-	-	-
	0.51	0.64	1.28	3.60	12.52	25.08	50.22	100.23	200.66	-
Local Attention	0.69	0.76	0.69	<u>0.80</u>	<u>2.04</u>	<u>3.88</u>	<u>7.67</u>	<u>15.04</u>	<u>30.11</u>	<u>63.15</u>
	1.24	2.49	4.77	9.42	18.65	37.12	74.15	148.35	-	-
Linformer	1.68	1.61	3.02	7.40	19.72	38.27	74.89	147.99	-	-
	1.24	1.25	2.04	2.91	6.78	19.67	-	-	-	-
Smyrf	1.27	1.23	1.24	1.85	4.99	10.21	24.89	-	-	-
	1.43	1.50	1.44	1.69	5.25	10.86	26.26	-	-	-
<b>FLASHATTENTION</b>	<b>0.11</b>	<b>0.16</b>	<b>0.52</b>	1.62	5.45	21.57	84.75	336.00	1338.56	5343.19
<b>Block-Sparse FLASHATTENTION</b>	0.11	0.12	0.16	<u>0.38</u>	<u>1.20</u>	<u>2.34</u>	<u>4.69</u>	<u>9.10</u>	<u>18.74</u>	<u>37.04</u>

Table 20: Forward pass + backward pass runtime (ms) of various exact/approximate/sparse attention mechanisms by sequence length. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	0.67	0.70	1.18	3.67	13.22	50.44	-	-	-	-
	0.74	<u>0.65</u>	1.23	3.80	13.21	-	-	-	-	-
Reformer	3.93	7.01	13.15	25.89	52.09	105.00	215.13	-	-	-
	1.09	1.27	1.99	5.38	18.32	36.77	73.67	147.29	296.35	-
Local Attention	1.31	1.25	1.30	<u>1.29</u>	<u>3.20</u>	<u>6.10</u>	<u>11.93</u>	<u>23.39</u>	<u>46.72</u>	<u>100.52</u>
	2.98	4.23	7.78	15.12	29.96	59.45	118.60	237.02	-	-
Linformer	3.03	3.05	4.26	10.70	30.77	60.15	118.33	234.94	-	-
	2.39	2.40	3.31	5.02	12.25	35.94	-	-	-	-
Smyrf	2.36	2.34	2.38	2.94	9.83	21.35	58.12	-	-	-
	2.35	2.35	2.37	3.25	10.36	22.57	60.63	-	-	-
<b>FLASHATTENTION</b>	<b>0.31</b>	<b>0.31</b>	<b>0.73</b>	2.29	7.64	30.09	118.50	470.51	1876.08	7492.85
<b>Block-Sparse FLASHATTENTION</b>	0.74	0.77	<u>0.82</u>	<b>0.88</b>	<b>1.71</b>	<b>3.21</b>	<b>6.56</b>	<b>12.60</b>	<b>24.93</b>	<b>50.39</b>

Table 21: Memory usage (MB) of various exact/approximate/sparse attention mechanisms by sequence length. Best in **bold**, second best underlined.

Attention Method	128	256	512	1024	2048	4096	8192	16384	32768	65536
PyTorch Attention	36	104	336	1184	4416	17024	-	-	-	-
	36	104	336	1184	4416	-	-	-	-	-
Reformer	377	754	1508	3016	6033	12067	24134	-	-	-
	53	110	232	592	1696	3392	6784	13568	27136	-
Local Attention	25	52	114	287	832	1652	3292	6572	13132	26252
	217	434	868	1737	3474	6947	13894	27788	-	-
Linformer	72	152	333	796	2540	5068	10125	20240	-	-
	33	82	228	408	910	2401	-	-	-	-
Smyrf	30	61	124	277	681	1370	2748	-	-	-
	33	66	131	294	708	1431	2872	-	-	-
<b>FLASHATTENTION</b>	<b>22</b>	<b>44</b>	<b>104</b>	<b>209</b>	<b>418</b>	<b>836</b>	<b>1672</b>	<b>3344</b>	<b>6688</b>	<b>13376</b>
<b>Block-Sparse FLASHATTENTION</b>	<b>22</b>	<b>44</b>	<u>104</u>	<b>209</b>	<b>418</b>	<b>836</b>	<b>1672</b>	<b>3344</b>	<b>6690</b>	<b>13384</b>

# FLASHATTENTION-2: Faster Attention with Better Parallelism and Work Partitioning

Tri Dao<sup>1,2</sup>

<sup>1</sup>Department of Computer Science, Princeton University

<sup>2</sup>Department of Computer Science, Stanford University

[trid@cs.stanford.edu](mailto:trid@cs.stanford.edu)

July 18, 2023

## Abstract

Scaling Transformers to longer sequence lengths has been a major problem in the last several years, promising to improve performance in language modeling and high-resolution image understanding, as well as to unlock new applications in code, audio, and video generation. The attention layer is the main bottleneck in scaling to longer sequences, as its runtime and memory increase quadratically in the sequence length. FLASHATTENTION [5] exploits the asymmetric GPU memory hierarchy to bring significant memory saving (linear instead of quadratic) and runtime speedup (2-4× compared to optimized baselines), with no approximation. However, FLASHATTENTION is still not nearly as fast as optimized matrix-multiply (GEMM) operations, reaching only 25-40% of the theoretical maximum FLOPs/s. We observe that the inefficiency is due to suboptimal work partitioning between different thread blocks and warps on the GPU, causing either low-occupancy or unnecessary shared memory reads/writes. We propose FLASHATTENTION-2, with better work partitioning to address these issues. In particular, we (1) tweak the algorithm to reduce the number of non-matmul FLOPs (2) parallelize the attention computation, even for a single head, across different thread blocks to increase occupancy, and (3) within each thread block, distribute the work between warps to reduce communication through shared memory. These yield around 2× speedup compared to FLASHATTENTION, reaching 50-73% of the theoretical maximum FLOPs/s on A100 and getting close to the efficiency of GEMM operations. We empirically validate that when used end-to-end to train GPT-style models, FLASHATTENTION-2 reaches training speed of up to 225 TFLOPs/s per A100 GPU (72% model FLOPs utilization).<sup>1</sup>

## 1 Introduction

Scaling up the context length of Transformers [18] is a challenge, since the attention layer at their heart has runtime and memory requirements quadratic in the input sequence length. Ideally, we would like to go beyond the standard 2k sequence length limit to train models to understand books, high resolution images, and long-form videos. Just within the last year, there have been several language models with much longer context than before: GPT-4 [12] with context length 32k, MosaicML’s MPT with context length 65k, and Anthropic’s Claude with context length 100k. Emerging use cases such as long document querying and story writing have demonstrated a need for models with such long context.

To reduce the computational requirement of attention on such long context, there have been numerous methods proposed to approximate attention [2, 3, 4, 8, 9, 14, 19, 20]. Though these methods have seen some use cases, as far as we know, most large-scale training runs still use standard attention. Motivated by this, Dao et al. [5] proposed to reorder the attention computation and leverages classical techniques (tiling, recomputation) to significantly speed it up and reduce memory usage from quadratic to linear in sequence length. This yields 2-4× wall-clock time speedup over optimized baselines, up to 10-20× memory saving,

---

<sup>1</sup>FLASHATTENTION-2 is available at <https://github.com/Dao-AI-Lab/flash-attention>

with no approximation, and as a result FLASHATTENTION has seen wide adoption in large-scale training and inference of Transformers.

However, context length increases even more, FLASHATTENTION is still not nearly as efficient as other primitives such as matrix-multiply (GEMM). In particular, while FLASHATTENTION is already 2-4 $\times$  faster than a standard attention implementation, the forward pass only reaches 30-50% of the theoretical maximum FLOPs/s of the device (Fig. 5), while the backward pass is even more challenging, reaching only 25-35% of maximum throughput on A100 GPU (Fig. 6). In contrast, optimized GEMM can reach up to 80-90% of the theoretical maximum device throughput. Through careful profiling, we observe that FLASHATTENTION still has suboptimal work partitioning between different thread blocks and warps on the GPU, causing either low-occupancy or unnecessary shared memory reads/writes.

Building on FLASHATTENTION, we propose FLASHATTENTION-2 with better parallelism and work partitioning to address these challenges.

1. In Section 3.1, we tweak the algorithms to reduce the number of non-matmul FLOPs while not changing the output. While the non-matmul FLOPs only account for a small fraction of the total FLOPs, they take longer to perform as GPUs have specialized units for matrix multiply, and as a result the matmul throughput can be up to 16 $\times$  higher than non-matmul throughput. It is thus important to reduce non-matmul FLOPs and spend as much time as possible doing matmul FLOPs.
2. We propose to parallelize both the forward pass and backward pass along the sequence length dimension, in addition to the batch and number of heads dimension. This increases occupancy (utilization of GPU resources) in the case where the sequences are long (and hence batch size is often small).
3. Even within one block of attention computation, we partition the work between different warps of a thread block to reduce communication and shared memory reads/writes.

In Section 4, we empirically validate that FLASHATTENTION-2 yields significant speedup compared to even FLASHATTENTION. Benchmarks on different settings (with or without causal mask, different head dimensions) show that FLASHATTENTION-2 achieves around 2 $\times$  speedup over FLASHATTENTION, reaching up to 73% of the theoretical max throughput in the forward pass, and up to 63% of the theoretical max throughput in the backward pass. When used end-to-end to train GPT-style models, we reach training speed of up to 225 TFLOPs/s per A100 GPU.

## 2 Background

We provide some background on the performance characteristics and execution model of GPUs. We also describe the standard implementation of attention, as well as FLASHATTENTION.

### 2.1 Hardware characteristics

**GPU performance characteristics.** The GPU consists of compute elements (e.g., floating point arithmetic units) and a memory hierarchy. Most modern GPUs contain specialized units to accelerate matrix multiply in low-precision (e.g., Tensor Cores on Nvidia GPUs for FP16/BF16 matrix multiply). The memory hierarchy comprise of high bandwidth memory (HBM), and on-chip SRAM (aka shared memory). As an example, the A100 GPU has 40-80GB of high bandwidth memory (HBM) with bandwidth 1.5-2.0TB/s and 192KB of on-chip SRAM per each of 108 streaming multiprocessors with bandwidth estimated around 19TB/s [6, 7]. As the L2 cache is not directly controllable by the programmer, we focus on the HBM and SRAM for the purpose of this discussion.

**Execution Model.** GPUs have a massive number of threads to execute an operation (called a kernel). Threads are organized into thread blocks, which are scheduled to run on streaming multiprocessors (SMs). Within each thread blocks, threads are grouped into warps (a group of 32 threads). Threads within a warp can communicate by fast shuffle instructions or cooperate to perform matrix multiply. Warps within a thread block can communicate by reading from / writing to shared memory. Each kernel loads inputs from HBM to registers and SRAM, computes, then writes outputs to HBM.

## 2.2 Standard Attention Implementation

Given input sequences  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$  where  $N$  is the sequence length and  $d$  is the head dimension, we want to compute the attention output  $\mathbf{O} \in \mathbb{R}^{N \times d}$ :

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \text{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d},$$

where softmax is applied row-wise.<sup>2</sup> For multi-head attention (MHA), this same computation is performed in parallel across many heads, and parallel over the batch dimension (number of input sequences in a batch).

The backward pass of attention proceeds as follows. Let  $\mathbf{dO} \in \mathbb{R}^{N \times d}$  be the gradient of  $\mathbf{O}$  with respect to some loss function. Then by the chain rule (aka backpropagation):

$$\begin{aligned} \mathbf{dV} &= \mathbf{P}^\top \mathbf{dO} \in \mathbb{R}^{N \times d} \\ \mathbf{dP} &= \mathbf{dO}\mathbf{V}^\top \in \mathbb{R}^{N \times N} \\ \mathbf{dS} &= \text{dsoftmax}(\mathbf{dP}) \in \mathbb{R}^{N \times N} \\ \mathbf{dQ} &= \mathbf{dS}\mathbf{K} \in \mathbb{R}^{N \times d} \\ \mathbf{dK} &= \mathbf{Q}\mathbf{dS}^\top \in \mathbb{R}^{N \times d}, \end{aligned}$$

where  $\text{dsoftmax}$  is the gradient (backward pass) of softmax applied row-wise. One can work out that if  $p = \text{softmax}(s)$  for some vector  $s$  and  $p$ , then with output gradient  $dp$ , the input gradient  $ds = (\text{diag}(p) - pp^\top)dp$ .

Standard attention implementations materialize the matrices  $\mathbf{S}$  and  $\mathbf{P}$  to HBM, which takes  $O(N^2)$  memory. Often  $N \gg d$  (typically  $N$  is on the order of 1k–8k and  $d$  is around 64–128). The standard attention implementation (1) calls the matrix multiply (GEMM) subroutine to multiply  $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$ , writes the result to HBM, then (2) loads  $\mathbf{S}$  from HBM to compute softmax and write the result  $\mathbf{P}$  to HBM, and finally (3) calls GEMM to get  $\mathbf{O} = \mathbf{P}\mathbf{V}$ . As most of the operations are bounded by memory bandwidth, the large number of memory accesses translates to slow wall-clock time. Moreover, the required memory is  $O(N^2)$  due to having to materialize  $\mathbf{S}$  and  $\mathbf{P}$ . Moreover, one has to save  $\mathbf{P} \in \mathbb{R}^{N \times N}$  for the backward pass to compute the gradients.

## 2.3 FLASHATTENTION

To speed up attention on hardware accelerators such as GPU, [5] proposes an algorithm to reduce the memory reads/writes while maintaining the same output (without approximation).

### 2.3.1 Forward pass

FLASHATTENTION applies the classical technique of tiling to reduce memory IOs, by (1) loading blocks of inputs from HBM to SRAM, (2) computing attention with respect to that block, and then (3) updating the output without writing the large intermediate matrices  $\mathbf{S}$  and  $\mathbf{P}$  to HBM. As the softmax couples entire rows or blocks of row, online softmax [11, 13] can split the attention computation into blocks, and rescale the output of each block to finally get the right result (with no approximation). By significantly reducing the amount of memory reads/writes, FLASHATTENTION yields 2–4× wall-clock speedup over optimized baseline attention implementations.

We describe the online softmax technique [11] and how it is used in attention [13]. For simplicity, consider just one row block of the attention matrix  $\mathbf{S}$ , of the form  $[\mathbf{S}^{(1)} \quad \mathbf{S}^{(2)}]$  for some matrices  $\mathbf{S}^{(1)}, \mathbf{S}^{(2)} \in \mathbb{R}^{B_r \times B_c}$ , where  $B_r$  and  $B_c$  are the row and column block sizes. We want to compute softmax of this row block and multiply with the value, of the form  $\begin{bmatrix} \mathbf{V}^{(1)} \\ \mathbf{V}^{(2)} \end{bmatrix}$  for some matrices  $\mathbf{V}^{(1)}, \mathbf{V}^{(2)} \in \mathbb{R}^{B_c \times d}$ . Standard softmax would

---

<sup>2</sup>For clarity of exposition, we omit the scaling of  $\mathbf{Q}\mathbf{K}^\top$  (typically by  $1/d$ ), and optionally elementwise masking on  $\mathbf{S}$  and/or dropout applied to  $\mathbf{P}$

compute:

$$\begin{aligned}
m &= \max(\text{rowmax}(\mathbf{S}^{(1)}), \text{rowmax}(\mathbf{S}^{(2)})) \in \mathbb{R}^{B_r} \\
\ell &= \text{rowsum}(e^{\mathbf{S}^{(1)} - m}) + \text{rowsum}(e^{\mathbf{S}^{(2)} - m}) \in \mathbb{R}^{B_r} \\
\mathbf{P} &= [\mathbf{P}^{(1)} \quad \mathbf{P}^{(2)}] = \text{diag}(\ell)^{-1} \begin{bmatrix} e^{\mathbf{S}^{(1)} - m} & e^{\mathbf{S}^{(2)} - m} \end{bmatrix} \in \mathbb{R}^{B_r \times 2B_c} \\
\mathbf{O} &= [\mathbf{P}^{(1)} \quad \mathbf{P}^{(2)}] \begin{bmatrix} \mathbf{V}^{(1)} \\ \mathbf{V}^{(2)} \end{bmatrix} = \text{diag}(\ell)^{-1} e^{\mathbf{S}^{(1)} - m} \mathbf{V}^{(1)} + e^{\mathbf{S}^{(2)} - m} \mathbf{V}^{(2)} \in \mathbb{R}^{B_r \times d}.
\end{aligned}$$

Online softmax instead computes “local” softmax with respect to each block and rescale to get the right output at the end:

$$\begin{aligned}
m^{(1)} &= \text{rowmax}(\mathbf{S}^{(1)}) \in \mathbb{R}^{B_r} \\
\ell^{(1)} &= \text{rowsum}(e^{\mathbf{S}^{(1)} - m^{(1)}}) \in \mathbb{R}^{B_r} \\
\tilde{\mathbf{P}}^{(1)} &= \text{diag}(\ell^{(1)})^{-1} e^{\mathbf{S}^{(1)} - m^{(1)}} \in \mathbb{R}^{B_r \times B_c} \\
\mathbf{O}^{(1)} &= \tilde{\mathbf{P}}^{(1)} \mathbf{V}^{(1)} = \text{diag}(\ell^{(1)})^{-1} e^{\mathbf{S}^{(1)} - m^{(1)}} \mathbf{V}^{(1)} \in \mathbb{R}^{B_r \times d} \\
m^{(2)} &= \max(m^{(1)}, \text{rowmax}(\mathbf{S}^{(2)})) = m \\
\ell^{(2)} &= e^{m^{(1)} - m^{(2)}} \ell^{(1)} + \text{rowsum}(e^{\mathbf{S}^{(2)} - m^{(2)}}) = \text{rowsum}(e^{\mathbf{S}^{(1)} - m}) + \text{rowsum}(e^{\mathbf{S}^{(2)} - m}) = \ell \\
\tilde{\mathbf{P}}^{(2)} &= \text{diag}(\ell^{(2)})^{-1} e^{\mathbf{S}^{(2)} - m^{(2)}} \\
\mathbf{O}^{(2)} &= \text{diag}(\ell^{(1)} / \ell^{(2)})^{-1} \mathbf{O}^{(1)} + \tilde{\mathbf{P}}^{(2)} \mathbf{V}^{(2)} = \text{diag}(\ell^{(2)})^{-1} e^{\mathbf{S}^{(1)} - m} \mathbf{V}^{(1)} + \text{diag}(\ell^{(2)})^{-1} e^{\mathbf{S}^{(2)} - m} \mathbf{V}^{(2)} = \mathbf{O}.
\end{aligned}$$

We show how FLASHATTENTION uses online softmax to enable tiling (Fig. 1) to reduce memory reads/writes.

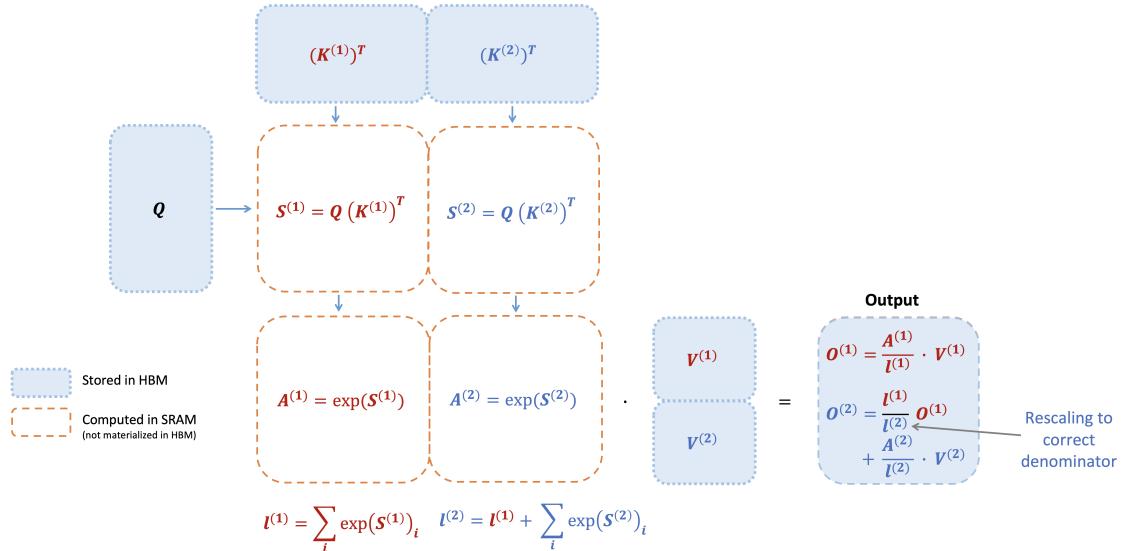


Figure 1: Diagram of how FLASHATTENTION forward pass is performed, when the key  $\mathbf{K}$  is partitioned into two blocks and the value  $\mathbf{V}$  is also partitioned into two blocks. By computing attention with respect to each block and rescaling the output, we get the right answer at the end, while avoiding expensive memory reads/writes of the intermediate matrices  $\mathbf{S}$  and  $\mathbf{P}$ . We simplify the diagram, omitting the step in softmax that subtracts each element by the row-wise max.

### 2.3.2 Backward pass

In the backward pass, by re-computing the values of the attention matrices  $\mathbf{S}$  and  $\mathbf{P}$  once blocks of inputs  $\mathbf{Q}, \mathbf{K}, \mathbf{V}$  are already loaded to SRAM, FLASHATTENTION avoids having to store large intermediate values. By not having to save the large matrices  $\mathbf{S}$  and  $\mathbf{P}$  of size  $N \times N$ , FLASHATTENTION yields 10-20 $\times$  memory saving depending on sequence length (memory required is linear in sequence length  $N$  instead of quadratic). The backward pass also achieves 2-4 $\times$  wall-clock speedup due to reduce memory reads/writes.

The backward pass applies tiling to the equations in Section 2.2. Though the backward pass is simpler than the forward pass conceptually (there is no softmax rescaling), the implementation is significantly more involved. This is because there are more values to be kept in SRAM to perform 5 matrix multiples in the backward pass, compared to just 2 matrix multiples in the forward pass.

## 3 FLASHATTENTION-2: Algorithm, Parallelism, and Work Partitioning

We describe the FLASHATTENTION-2 algorithm, which includes several tweaks to FLASHATTENTION to reduce the number of non-matmul FLOPs. We then describe how to parallelize the computation on different thread blocks to make full use of the GPU resources. Finally we describe we partition the work between different warps within one thread block to reduce the amount of shared memory access. These improvements lead to 2-3 $\times$  speedup as validated in Section 4.

### 3.1 Algorithm

We tweak the algorithm from FLASHATTENTION to reduce the number of non-matmul FLOPs. This is because modern GPUs have specialized compute units (e.g., Tensor Cores on Nvidia GPUs) that makes matmul much faster. As an example, the A100 GPU has a max theoretical throughput of 312 TFLOPs/s of FP16/BF16 matmul, but only 19.5 TFLOPs/s of non-matmul FP32. Another way to think about this is that each non-matmul FLOP is 16 $\times$  more expensive than a matmul FLOP. To maintain high throughput (e.g., more than 50% of the maximum theoretical TFLOPs/s), we want to spend as much time on matmul FLOPs as possible.

#### 3.1.1 Forward pass

We revisit the online softmax trick as shown in Section 2.3 and make two minor tweaks to reduce non-matmul FLOPs:

1. We do not have to rescale both terms of the output update by  $\text{diag}(\ell^{(2)})^{-1}$ :

$$\mathbf{O}^{(2)} = \text{diag}(\ell^{(1)} / \ell^{(2)})^{-1} \mathbf{O}^{(1)} + \text{diag}(\ell^{(2)})^{-1} e^{\mathbf{S}^{(2)} - \mathbf{m}^{(2)}} \mathbf{V}^{(2)}.$$

We can instead maintain an “un-scaled” version of  $\mathbf{O}^{(2)}$  and keep around the statistics  $\ell^{(2)}$ :

$$\tilde{\mathbf{O}}^{(2)} = \text{diag}(\ell^{(1)})^{-1} \mathbf{O}^{(1)} + e^{\mathbf{S}^{(2)} - \mathbf{m}^{(2)}} \mathbf{V}^{(2)}.$$

Only at the every end of the loop do we scale the final  $\tilde{\mathbf{O}}^{(\text{last})}$  by  $\text{diag}(\ell^{(\text{last})})^{-1}$  to get the right output.

2. We do not have to save both the max  $\mathbf{m}^{(j)}$  and the sum of exponentials  $\ell^{(j)}$  for the backward pass. We only need to store the logsumexp  $L^{(j)} = \mathbf{m}^{(j)} + \log(\ell^{(j)})$ .

In the simple case of 2 blocks in Section 2.3, the online softmax trick now becomes:

$$\begin{aligned}
m^{(1)} &= \text{rowmax}(\mathbf{S}^{(1)}) \in \mathbb{R}^{B_r} \\
\ell^{(1)} &= \text{rowsum}(e^{\mathbf{S}^{(1)} - m^{(1)}}) \in \mathbb{R}^{B_r} \\
\tilde{\mathbf{O}}^{(1)} &= e^{\mathbf{S}^{(1)} - m^{(1)}} \mathbf{V}^{(1)} \in \mathbb{R}^{B_r \times d} \\
m^{(2)} &= \max(m^{(1)}, \text{rowmax}(\mathbf{S}^{(2)})) = m \\
\ell^{(2)} &= e^{m^{(1)} - m^{(2)}} \ell^{(1)} + \text{rowsum}(e^{\mathbf{S}^{(2)} - m^{(2)}}) = \text{rowsum}(e^{\mathbf{S}^{(1)} - m}) + \text{rowsum}(e^{\mathbf{S}^{(2)} - m}) = \ell \\
\tilde{\mathbf{P}}^{(2)} &= \text{diag}(\ell^{(2)})^{-1} e^{\mathbf{S}^{(2)} - m^{(2)}} \\
\tilde{\mathbf{O}}^{(2)} &= \text{diag}(e^{m^{(1)} - m^{(2)}}) \tilde{\mathbf{O}}^{(1)} + e^{\mathbf{S}^{(2)} - m^{(2)}} \mathbf{V}^{(2)} = e^{s^{(1)} - m} \mathbf{V}^{(1)} + e^{s^{(2)} - m} \mathbf{V}^{(2)} \\
\mathbf{O}^{(2)} &= \text{diag}(\ell^{(2)})^{-1} \tilde{\mathbf{O}}^{(2)} = \mathbf{O}.
\end{aligned}$$

We describe the full FLASHATTENTION-2 forward pass in Algorithm 1.

---

**Algorithm 1** FLASHATTENTION-2 forward pass

---

**Require:** Matrices  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$  in HBM, block sizes  $B_c, B_r$ .

- 1: Divide  $\mathbf{Q}$  into  $T_r = \left\lceil \frac{N}{B_r} \right\rceil$  blocks  $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$  of size  $B_r \times d$  each, and divide  $\mathbf{K}, \mathbf{V}$  into  $T_c = \left\lceil \frac{N}{B_c} \right\rceil$  blocks  $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$  and  $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$ , of size  $B_c \times d$  each.
  - 2: Divide the output  $\mathbf{O} \in \mathbb{R}^{N \times d}$  into  $T_r$  blocks  $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$  of size  $B_r \times d$  each, and divide the logsumexp  $L$  into  $T_r$  blocks  $L_i, \dots, L_{T_r}$  of size  $B_r$  each.
  - 3: **for**  $1 \leq i \leq T_r$  **do**
  - 4:   Load  $\mathbf{Q}_i$  from HBM to on-chip SRAM.
  - 5:   On chip, initialize  $\mathbf{O}_i^{(0)} = (0)_{B_r \times d} \in \mathbb{R}^{B_r \times d}, \ell_i^{(0)} = (0)_{B_r} \in \mathbb{R}^{B_r}, m_i^{(0)} = (-\infty)_{B_r} \in \mathbb{R}^{B_r}$ .
  - 6:   **for**  $1 \leq j \leq T_c$  **do**
  - 7:     Load  $\mathbf{K}_j, \mathbf{V}_j$  from HBM to on-chip SRAM.
  - 8:     On chip, compute  $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$ .
  - 9:     On chip, compute  $m_i^{(j)} = \max(m_i^{(j-1)}, \text{rowmax}(\mathbf{S}_i^{(j)})) \in \mathbb{R}^{B_r}, \tilde{\mathbf{P}}_i^{(j)} = \exp(\mathbf{S}_i^{(j)} - m_i^{(j)}) \in \mathbb{R}^{B_r \times B_c}$  (pointwise),  $\ell_i^{(j)} = e^{m_i^{(j-1)} - m_i^{(j)}} \ell_i^{(j-1)} + \text{rowsum}(\tilde{\mathbf{P}}_i^{(j)}) \in \mathbb{R}^{B_r}$ .
  - 10:    On chip, compute  $\mathbf{O}_i^{(j)} = \text{diag}(e^{m_i^{(j-1)} - m_i^{(j)}}) \mathbf{O}_i^{(j-1)} + \tilde{\mathbf{P}}_i^{(j)} \mathbf{V}_j$ .
  - 11:   **end for**
  - 12:   On chip, compute  $\mathbf{O}_i = \text{diag}(\ell_i^{(T_c)})^{-1} \mathbf{O}_i^{(T_c)}$ .
  - 13:   On chip, compute  $L_i = m_i^{(T_c)} + \log(\ell_i^{(T_c)})$ .
  - 14:   Write  $\mathbf{O}_i$  to HBM as the  $i$ -th block of  $\mathbf{O}$ .
  - 15:   Write  $L_i$  to HBM as the  $i$ -th block of  $L$ .
  - 16: **end for**
  - 17: Return the output  $\mathbf{O}$  and the logsumexp  $L$ .
- 

**Causal masking.** One common use case of attention is in auto-regressive language modeling, where we need to apply a causal mask to the attention matrix  $\mathbf{S}$  (i.e., any entry  $\mathbf{S}_{ij}$  with  $j > i$  is set to  $-\infty$ ).

1. As FLASHATTENTION and FLASHATTENTION-2 already operate by blocks, for any blocks where all the column indices are more than the row indices (approximately half of the blocks for large sequence length), we can skip the computation of that block. This leads to around 1.7-1.8× speedup compared to attention without the causal mask.
2. We do not need to apply the causal mask for blocks whose row indices are guaranteed to be strictly less than the column indices. This means that for each row, we only need apply causal mask to 1 block (assuming square block).

**Correctness, runtime, and memory requirement.** As with FLASHATTENTION, Algorithm 1 returns the correct output  $\mathbf{O} = \text{softmax}(\mathbf{Q}\mathbf{K}^\top)\mathbf{V}$  (with no approximation), using  $O(N^2d)$  FLOPs and requires  $O(N)$  additional memory beyond inputs and output (to store the logsumexp  $L$ ). The proof is almost the same as the proof of Dao et al. [5, Theorem 1], so we omit it here.

### 3.1.2 Backward pass

The backward pass of FLASHATTENTION-2 is almost the same as that of FLASHATTENTION. We make a minor tweak to only use the row-wise logsumexp  $L$  instead of both the row-wise max and row-wise sum of exponentials in the softmax. We include the backward pass description in Algorithm 2 for completeness.

---

**Algorithm 2** FLASHATTENTION-2 Backward Pass

---

**Require:** Matrices  $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}, \mathbf{dO} \in \mathbb{R}^{N \times d}$  in HBM, vector  $L \in \mathbb{R}^N$  in HBM, block sizes  $B_c, B_r$ .

- 1: Divide  $\mathbf{Q}$  into  $T_r = \left\lceil \frac{N}{B_r} \right\rceil$  blocks  $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$  of size  $B_r \times d$  each, and divide  $\mathbf{K}, \mathbf{V}$  in to  $T_c = \left\lceil \frac{N}{B_c} \right\rceil$  blocks  $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$  and  $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$ , of size  $B_c \times d$  each.
- 2: Divide  $\mathbf{O}$  into  $T_r$  blocks  $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$  of size  $B_r \times d$  each, divide  $\mathbf{dO}$  into  $T_r$  blocks  $\mathbf{dO}_i, \dots, \mathbf{dO}_{T_r}$  of size  $B_r \times d$  each, and divide  $L$  into  $T_r$  blocks  $L_i, \dots, L_{T_r}$  of size  $B_r$  each.
- 3: Initialize  $\mathbf{dQ} = (0)_{N \times d}$  in HBM and divide it into  $T_r$  blocks  $\mathbf{dQ}_1, \dots, \mathbf{dQ}_{T_r}$  of size  $B_r \times d$  each. Divide  $\mathbf{dK}, \mathbf{dV} \in \mathbb{R}^{N \times d}$  in to  $T_c$  blocks  $\mathbf{dK}_1, \dots, \mathbf{dK}_{T_c}$  and  $\mathbf{dV}_1, \dots, \mathbf{dV}_{T_c}$ , of size  $B_c \times d$  each.
- 4: Compute  $D = \text{rowsum}(\mathbf{dO} \circ \mathbf{O}) \in \mathbb{R}^d$  (pointwise multiply), write  $D$  to HBM and divide it into  $T_r$  blocks  $D_1, \dots, D_{T_r}$  of size  $B_r$  each.
- 5: **for**  $1 \leq j \leq T_c$  **do**
- 6:   Load  $\mathbf{K}_j, \mathbf{V}_j$  from HBM to on-chip SRAM.
- 7:   Initialize  $\mathbf{dK}_j = (0)_{B_c \times d}, \mathbf{dV}_j = (0)_{B_c \times d}$  on SRAM.
- 8:   **for**  $1 \leq i \leq T_r$  **do**
- 9:     Load  $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, L_i, D_i$  from HBM to on-chip SRAM.
- 10:    On chip, compute  $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$ .
- 11:    On chip, compute  $\mathbf{P}_i^{(j)} = \exp(\mathbf{S}_i^{(j)} - L_i) \in \mathbb{R}^{B_r \times B_c}$ .
- 12:    On chip, compute  $\mathbf{dV}_j \leftarrow \mathbf{dV}_j + (\mathbf{P}_i^{(j)})^\top \mathbf{dO}_i \in \mathbb{R}^{B_c \times d}$ .
- 13:    On chip, compute  $\mathbf{dP}_i^{(j)} = \mathbf{dO}_i \mathbf{V}_j^\top \in \mathbb{R}^{B_r \times B_c}$ .
- 14:    On chip, compute  $\mathbf{dS}_i^{(j)} = \mathbf{P}_i^{(j)} \circ (\mathbf{dP}_i^{(j)} - D_i) \in \mathbb{R}^{B_r \times B_c}$ .
- 15:    Load  $\mathbf{dQ}_i$  from HBM to SRAM, then on chip, update  $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \mathbf{dS}_i^{(j)} \mathbf{K}_j \in \mathbb{R}^{B_r \times d}$ , and write back to HBM.
- 16:    On chip, compute  $\mathbf{dK}_j \leftarrow \mathbf{dK}_j + \mathbf{dS}_i^{(j)^\top} \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$ .
- 17:   **end for**
- 18:   Write  $\mathbf{dK}_j, \mathbf{dV}_j$  to HBM.
- 19: **end for**
- 20: Return  $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$ .

---

**Multi-query attention and grouped-query attention.** Multi-query attention (MQA) [15] and grouped-query attention (GQA) [1] are variants of attention where multiple heads of query attend to the same head of key and value, in order to reduce the size of KV cache during inference. Instead of having to duplicate the key and value heads for the computation, we implicitly manipulate the indices into the head to perform the same computation. In the backward pass, we need to sum the gradients  $\mathbf{dK}$  and  $\mathbf{dV}$  across different heads that were implicitly duplicated.

## 3.2 Parallelism

The first version of FLASHATTENTION parallelizes over batch size and number of heads. We use 1 thread block to process one attention head, and there are overall batch size · number of heads thread blocks. Each thread block is scheduled to run on a streaming multiprocessor (SM), and there are 108 of these SMs on

an A100 GPU for example. This scheduling is efficient when this number is large (say  $\geq 80$ ), since we can effectively use almost all of the compute resources on the GPU.

In the case of long sequences (which usually means small batch sizes or small number of heads), to make better use of the multiprocessors on the GPU, we now additionally parallelize over the sequence length dimension. This results in significant speedup for this regime.

**Forward pass.** We see that the outer loop (over sequence length) is embarrassingly parallel, and we schedule them on different thread blocks that do not need to communicate with each other. We also parallelize over the batch dimension and number of heads dimension, as done in FLASHATTENTION. The increased parallelism over sequence length helps improve occupancy (fraction of GPU resources being used) when the batch size and number of heads are small, leading to speedup in this case.

These ideas of swapping the order of the loop (outer loop over row blocks and inner loop over column blocks, instead of the other way round in the original FLASHATTENTION paper), as well as parallelizing over the sequence length dimension were first suggested and implemented by Phil Tillet in the Triton [17] implementation.<sup>3</sup>

**Backward pass.** Notice that the only shared computation between different column blocks is in update  $\mathbf{dQ}$  in Algorithm 2, where we need to load  $\mathbf{dQ}_i$  from HBM to SRAM, then on chip, update  $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \mathbf{dS}_i^{(j)} \mathbf{K}_j$ , and write back to HBM. We thus parallelize over the sequence length dimension as well, and schedule 1 thread block for each column block of the backward pass. We use atomic adds to communicate between different thread blocks to update  $\mathbf{dQ}$ .

We describe the parallelization scheme in Fig. 2.

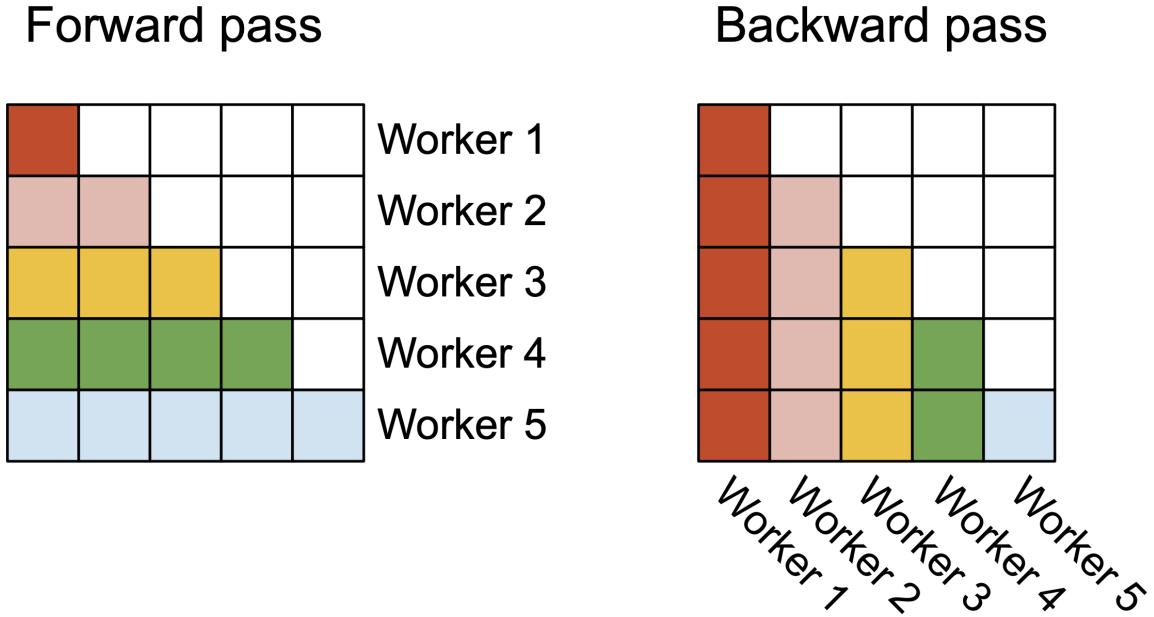


Figure 2: In the forward pass (left), we parallelize the workers (thread blocks) where each worker takes care of a block of rows of the attention matrix. In the backward pass (right), each worker takes care of a block of columns of the attention matrix.

---

<sup>3</sup><https://github.com/openai/triton/blob/main/python/tutorials/06-fused-attention.py>

### 3.3 Work Partitioning Between Warps

As Section 3.2 describe how we schedule thread blocks, even within each thread block, we also have to decide how to partition the work between different warps. We typically use 4 or 8 warps per thread block, and the partitioning is described in Fig. 3.

**Forward pass.** For each block, FLASHATTENTION splits  $\mathbf{K}$  and  $\mathbf{V}$  across 4 warps while keeping  $\mathbf{Q}$  accessible by all warps. Each warp multiplies to get a slice of  $\mathbf{QK}^T$ , then they need to multiply with a slice of  $\mathbf{V}$  and communicate to add up the result. This is referred to as the “split-K” scheme. However, this is inefficient since all warps need to write their intermediate results out to shared memory, synchronize, then add up the intermediate results. These shared memory reads/writes slow down the forward pass in FLASHATTENTION.

In FLASHATTENTION-2, we instead split  $\mathbf{Q}$  across 4 warps while keeping  $\mathbf{K}$  and  $\mathbf{V}$  accessible by all warps. After each warp performs matrix multiply to get a slice of  $\mathbf{QK}^T$ , they just need to multiply with their shared slice of  $\mathbf{V}$  to get their corresponding slice of the output. There is no need for communication between warps. The reduction in shared memory reads/writes yields speedup (Section 4).

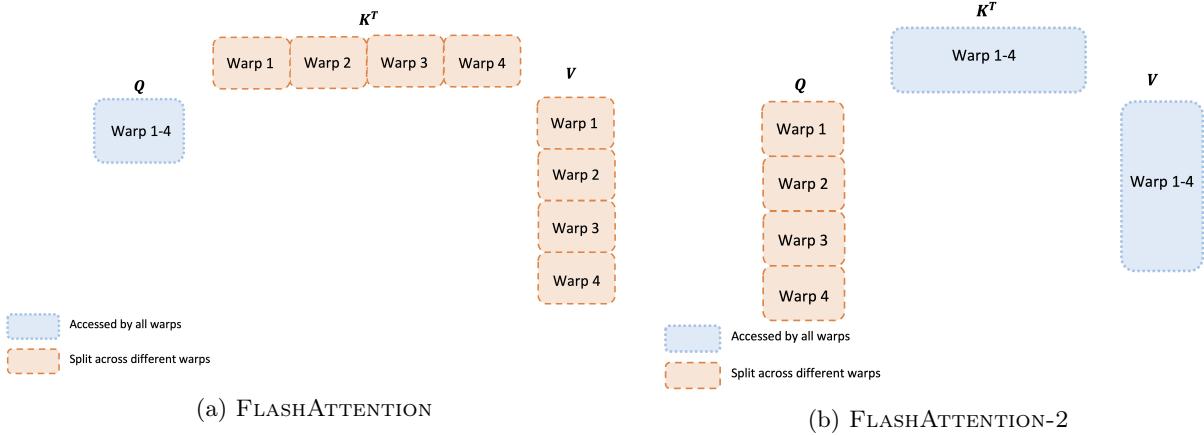


Figure 3: Work partitioning between different warps in the forward pass

**Backward pass.** Similarly for the backward pass, we choose to partition the warps to avoid the “split-K” scheme. However, it still requires some synchronization due to the more complicated dependency between all the different inputs and gradients  $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}, \mathbf{dO}, \mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$ . Nevertheless, avoiding “split-K” reduces shared memory reads/writes and again yields speedup (Section 4).

**Tuning block sizes** Increasing block sizes generally reduces shared memory loads/stores, but increases the number of registers required and the total amount of shared memory. Past a certain block size, register spilling causes significant slowdown, or the amount of shared memory required is larger than what the GPU has available, and the kernel cannot run at all. Typically we choose blocks of size  $\{64, 128\} \times \{64, 128\}$ , depending on the head dimension  $d$  and the device shared memory size.

We manually tune for each head dimensions since there are essentially only 4 choices for block sizes, but this could benefit from auto-tuning to avoid this manual labor. We leave this to future work.

## 4 Empirical Validation

We evaluate the impact of using FLASHATTENTION-2 to train Transformer models.

- **Benchmarking attention.** We measure the runtime of FLASHATTENTION-2 across different sequence lengths and compare it to a standard implementation in PyTorch, FLASHATTENTION, and FLASHATTENTION in Triton. We confirm that FLASHATTENTION-2 is 1.7-3.0 $\times$  faster than FLASHATTENTION, 1.3-2.5 $\times$  faster than FLASHATTENTION in Triton, and 3-10 $\times$  faster than a standard attention implementation.

FLASHATTENTION-2 reaches up to 230 TFLOPs/s, 73% of the theoretical maximum TFLOPs/s on A100 GPUs.

- **End-to-end training speed** When used end-to-end to train GPT-style models of size 1.3B and 2.7B on sequence lengths either 2k or 8k, FLASHATTENTION-2 yields up to 1.3 $\times$  speedup compared to FLASHATTENTION and 2.8 $\times$  speedup compared to a baseline without FLASHATTENTION. FLASHATTENTION-2 reaches up to 225 TFLOPs/s (72% model FLOPs utilization) per A100 GPU.

## 4.1 Benchmarking Attention

We measure the runtime of different attention methods on an A100 80GB SXM4 GPU for different settings (without / with causal mask, head dimension 64 or 128). We report the results in Fig. 4, Fig. 5 and Fig. 6, showing that FLASHATTENTION-2 is around 2 $\times$  faster than FLASHATTENTION and FLASHATTENTION in **xformers** (the “cutlass” implementation). FLASHATTENTION-2 is around 1.3–1.5 $\times$  faster than FLASHATTENTION in Triton in the forward pass and around 2 $\times$  faster in the backward pass. Compared to a standard attention implementation in PyTorch, FLASHATTENTION-2 can be up to 10 $\times$  faster.

Benchmark setting: we vary the sequence length from 512, 1k, ..., 16k, and set batch size so that the total number of tokens is 16k. We set hidden dimension to 2048, and head dimension to be either 64 or 128 (i.e., 32 heads or 16 heads). To calculate the FLOPs of the forward pass, we use:

$$4 \cdot \text{seqlen}^2 \cdot \text{head dimension} \cdot \text{number of heads.}$$

With causal mask, we divide this number by 2 to account for the fact that approximately only half of the entries are calculated. To get the FLOPs of the backward pass, we multiply the forward pass FLOPs by 2.5 (since there are 2 matmuls in the forward pass and 5 matmuls in the backward pass, due to recomputation).

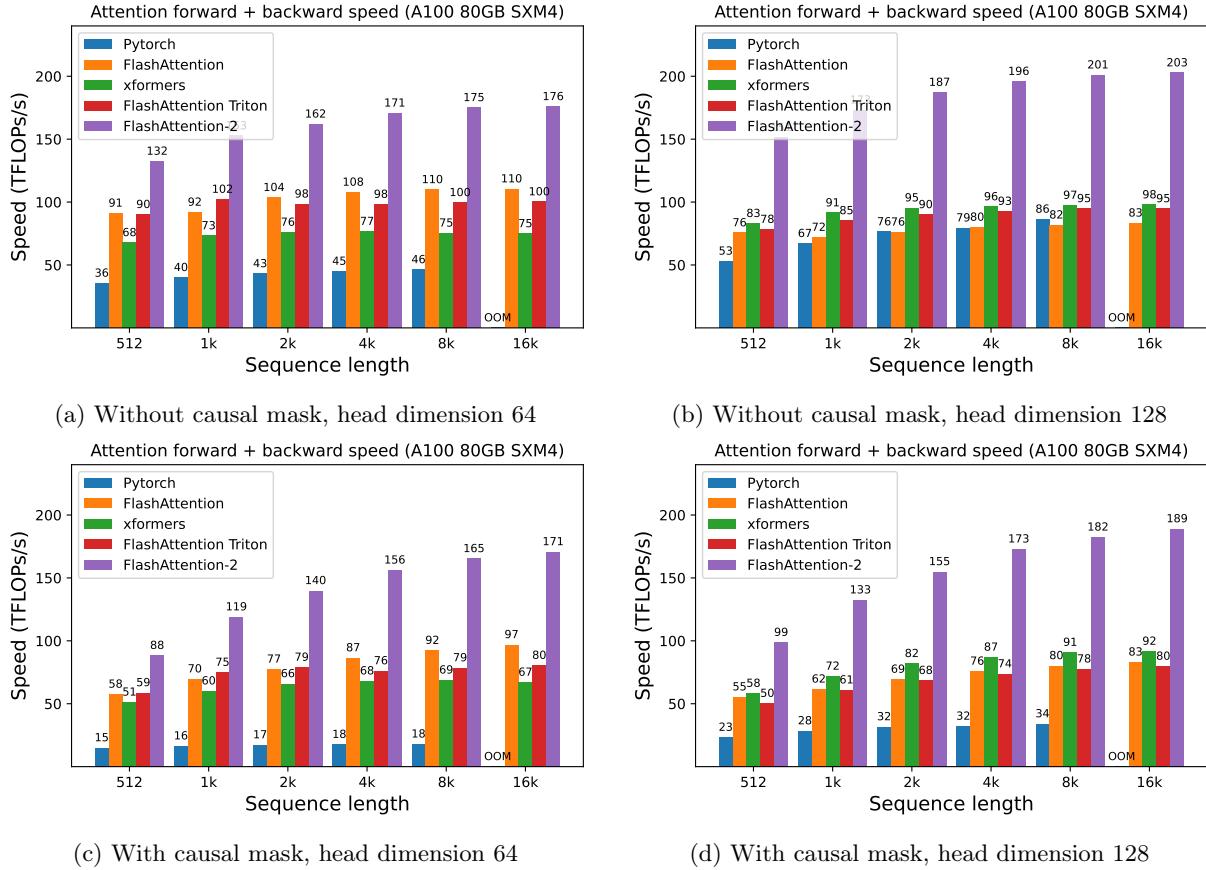


Figure 4: Attention forward + backward speed on A100 GPU

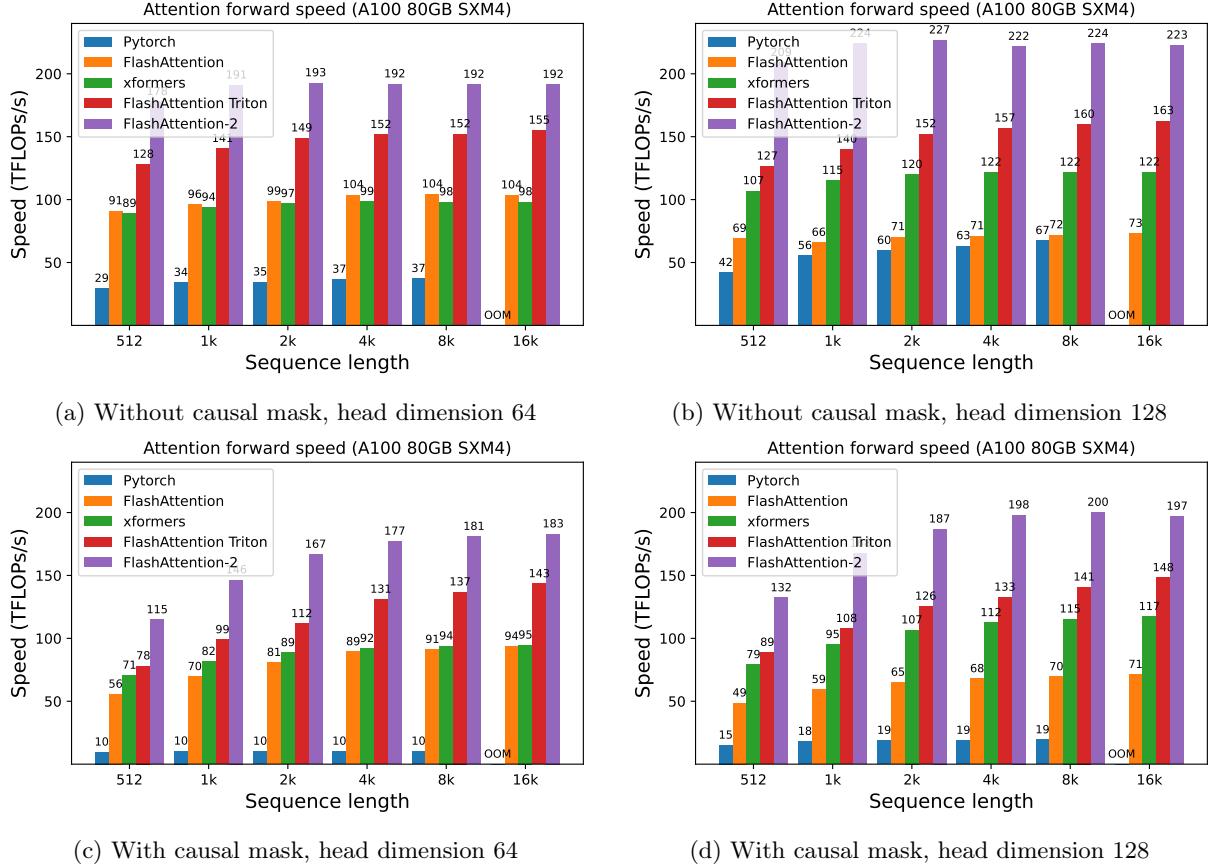


Figure 5: Attention forward speed on A100 GPU

Just running the same implementation on H100 GPUs (using no special instructions to make use of new features such as TMA and 4th-gen Tensor Cores), we obtain up to 335 TFLOPs/s (Fig. 7). We expect that by using new instructions, we can obtain another 1.5x-2x speedup on H100 GPUs. We leave that to future work.

## 4.2 End-to-end Performance

We measure the training throughput of GPT-style models with either 1.3B or 2.7B parameters, on 8×A100 80GB SXM4. As shown in Table 1, FLASHATTENTION-2 yields 2.8× speedup compared to a baseline without FLASHATTENTION and 1.3× speedup compared to FLASHATTENTION, reaching up to 225 TFLOPs/s per A100 GPU.

Note that we calculate the FLOPs by the formula, following Megatron-LM [16] (and many other papers and libraries):

$$6 \cdot \text{seqlen} \cdot \text{number of params} + 12 \cdot \text{number of layers} \cdot \text{hidden dim} \cdot \text{seqlen}^2.$$

The first term accounts for the FLOPs due to weight–input multiplication, and the second term accounts for the FLOPs due to attention. However, one can argue that the second term should be halved, as with causal mask we only need to compute approximately half the number of elements in attention. We choose to follow the formula from the literature (without dividing the attention FLOPs by 2) for consistency.

## 5 Discussion and Future Directions

FLASHATTENTION-2 is 2× faster than FLASHATTENTION, which means that we can train models with 16k longer context for the same price as previously training a 8k context model, for the same number of tokens.

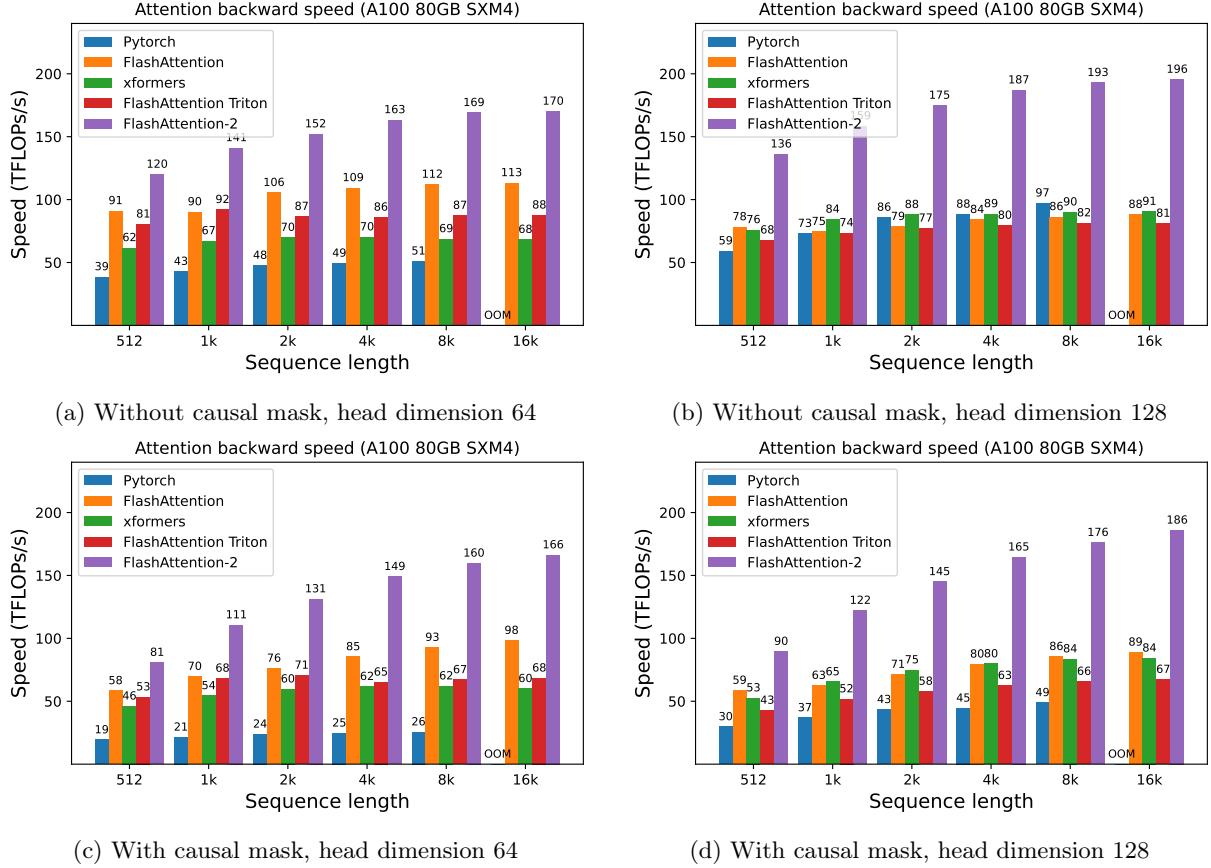


Figure 6: Attention backward speed on A100 GPU

Table 1: Training speed (TFLOPs/s/GPU) of GPT-style models on 8×A100 GPUs. FLASHATTENTION-2 reaches up to 225 TFLOPs/s (72% model FLOPs utilization). We compare against a baseline running without FLASHATTENTION.

Model	Without FLASHATTENTION	FLASHATTENTION	FLASHATTENTION-2
GPT3-1.3B 2k context	142 TFLOPs/s	189 TFLOPs/s	196 TFLOPs/s
GPT3-1.3B 8k context	72 TFLOPs/s	170 TFLOPs/s	220 TFLOPs/s
GPT3-2.7B 2k context	149 TFLOPs/s	189 TFLOPs/s	205 TFLOPs/s
GPT3-2.7B 8k context	80 TFLOPs/s	175 TFLOPs/s	225 TFLOPs/s

We are excited about how this can be used to understand long books and reports, high resolution images, audio and video. FLASHATTENTION-2 will also speed up training, finetuning, and inference of existing models.

In the near future, we plan to collaborate with researchers and engineers to make FlashAttention widely applicable in different kinds of devices (e.g., H100 GPUs, AMD GPUs), as well as new data types such as FP8. As an immediate next step, we plan to optimize FlashAttention-2 for H100 GPUs to use new hardware features (TMA, 4th-gen Tensor Cores, fp8). Combining the low-level optimizations in FlashAttention-2 with high-level algorithmic changes (e.g., local, dilated, block-sparse attention) could allow us to train AI models with much longer context. We are also excited to work with compiler researchers to make these optimization techniques easily programmable.

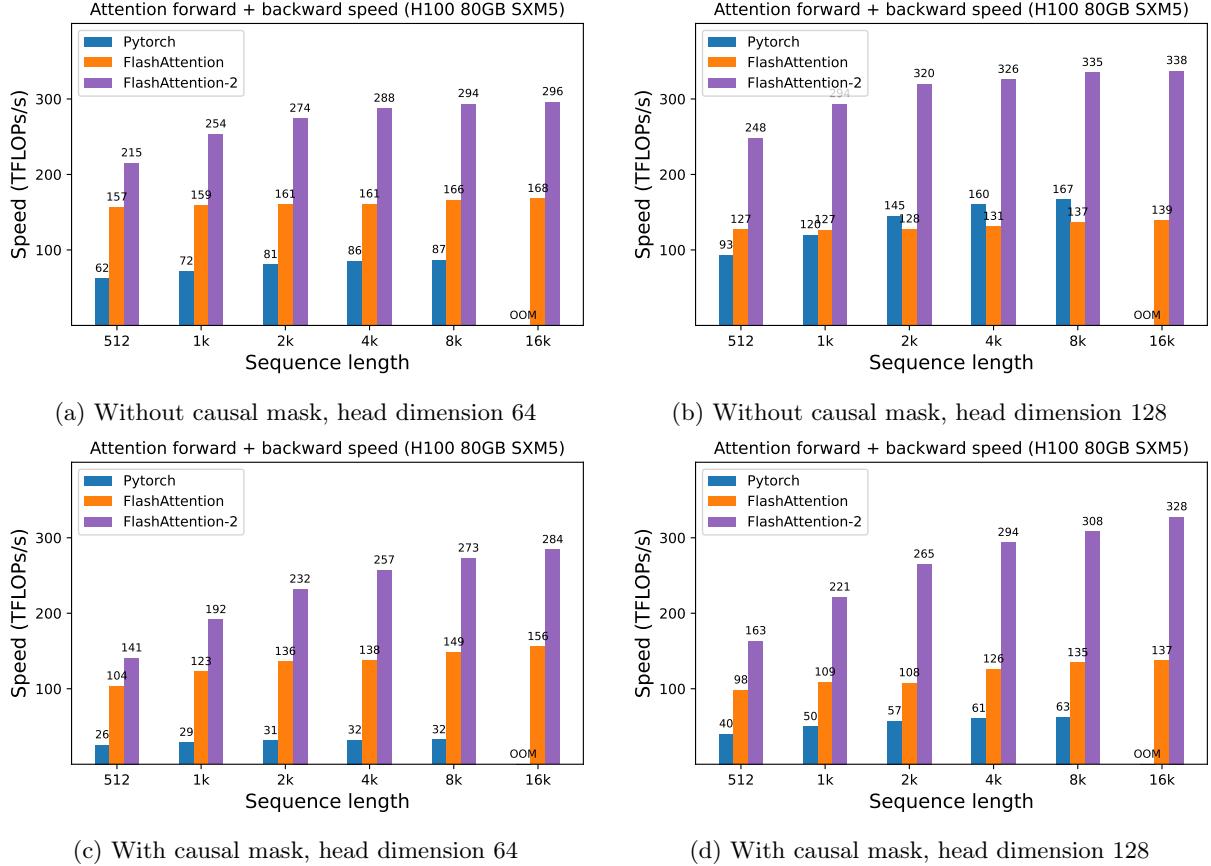


Figure 7: Attention forward + backward speed on H100 GPU

## Acknowledgments

We thank Phil Tillet and Daniel Haziza, who have implemented versions of FLASHATTENTION in Triton [17] and the `xformers` library [10]. FLASHATTENTION-2 was motivated by exchange of ideas between different ways that attention could be implemented. We are grateful to the Nvidia CUTLASS team (especially Vijay Thakkar, Cris Cecka, Haicheng Wu, and Andrew Kerr) for their CUTLASS library, in particular the CUTLASS 3.x release, which provides clean abstractions and powerful building blocks for the implementation of FLASHATTENTION-2. We thank Driss Guessous for integrating FLASHATTENTION to PyTorch. FLASHATTENTION-2 has benefited from helpful discussions with Phil Wang, Markus Rabe, James Bradbury, Young-Jun Ko, Julien Launay, Daniel Hesslow, Michaël Benesty, Horace He, Ashish Vaswani, and Erich Elsen. Thanks to Stanford CRFM and Stanford NLP for the compute support. We thank Dan Fu and Christopher Ré for their collaboration, constructive feedback, and constant encouragement on this line of work of designing hardware-efficient algorithms. We thank Albert Gu and Beidi Chen for their helpful suggestions on early drafts of this technical report.

## References

- [1] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.
- [2] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

- [3] Beidi Chen, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra, and Christopher Ré. Scatterbrain: Unifying sparse and low-rank attention. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.
- [4] Krzysztof Marcin Choromanski, Valerii Likhoshesterov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. In *International Conference on Learning Representations (ICLR)*, 2020.
- [5] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.
- [6] Zhe Jia and Peter Van Sandt. Dissecting the Ampere GPU architecture via microbenchmarking. GPU Technology Conference, 2021.
- [7] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia Volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.
- [8] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, pages 5156–5165. PMLR, 2020.
- [9] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *The International Conference on Machine Learning (ICML)*, 2020.
- [10] Benjamin Leflaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, and Daniel Haziza. xformers: A modular and hackable transformer modelling library. <https://github.com/facebookresearch/xformers>, 2022.
- [11] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.
- [12] OpenAI. Gpt-4 technical report. *ArXiv*, abs/2303.08774, 2023.
- [13] Markus N Rabe and Charles Staats. Self-attention does not need  $O(n^2)$  memory. *arXiv preprint arXiv:2112.05682*, 2021.
- [14] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9: 53–68, 2021.
- [15] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.
- [16] Mohammad Shoeybi, Mostafa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [17] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.
- [18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [19] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.
- [20] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in Neural Information Processing Systems*, 33, 2020.

---

# FLASHDECODING++: FASTER LARGE LANGUAGE MODEL INFERENCE ON GPUS

---

**Ke Hong<sup>†</sup>**  
Tsinghua University  
& Infinigence-AI

**Qiuli Mao**  
Tsinghua University  
& Infinigence-AI

**Kangdi Chen**  
Infinigence-AI

**Guohao Dai<sup>‡</sup>**  
Shanghai Jiao Tong University  
& Infinigence-AI

**XiuHong Li**  
Peking University

**Yuhan Dong**  
Tsinghua University

**Jiaming Xu<sup>†</sup>**  
Shanghai Jiao Tong University  
& Infinigence-AI

**Jun Liu**  
Shanghai Jiao Tong University  
& Infinigence-AI

**Yu Wang<sup>‡</sup>**  
Tsinghua University

<sup>†</sup>daiguohao@sjtu.edu.cn, daiguohao@infini-ai.com, yu-wang@tsinghua.edu.cn

## ABSTRACT

As the Large Language Model (LLM) becomes increasingly important in various domains, the performance of LLM inference is crucial to massive LLM applications. However, the following challenges still remain unsolved in accelerating LLM inference: (1) Synchronized partial softmax update. The softmax operation requires a synchronized update operation among each partial softmax result, leading to  $\sim 20\%$  overheads for the attention computation in LLMs. (2) Under-utilized computation of flat GEMM. The shape of matrices performing GEMM in LLM inference is flat, leading to under-utilized computation and  $>50\%$  performance loss after padding zeros in previous designs (*e.g.*, cuBLAS, CUTLASS, etc.). (3) Performance loss due to static dataflow. Kernel performance in LLM depends on varied input data features, hardware configurations, etc. A single and static dataflow may lead to a 50.25% performance loss for GEMMs of different shapes in LLM inference.

We present *FlashDecoding++*, a fast LLM inference engine supporting mainstream LLMs and hardware back-ends. To tackle the above challenges, *FlashDecoding++* creatively proposes: (1) **Asynchronous softmax with unified max value.** *FlashDecoding++* introduces a unified max value technique for different partial softmax computations to avoid synchronization. Based on this, the fine-grained pipelining is proposed. (2) **Flat GEMM optimization with double buffering.** *FlashDecoding++* points out that flat GEMMs with different shapes face varied bottlenecks. Then, techniques like double buffering are introduced. (3) **Heuristic dataflow with hardware resource adaptation.** *FlashDecoding++* heuristically optimizes dataflow using different hardware resource (*e.g.*, Tensor Core or CUDA core) considering input dynamics. Due to the versatility of optimizations in *FlashDecoding++*, *FlashDecoding++* can achieve up to **4.86 $\times$**  and **2.18 $\times$**  speedup on both NVIDIA and AMD GPUs compared to Hugging Face implementations. *FlashDecoding++* also achieves an average speedup of **1.37 $\times$**  compared to state-of-the-art LLM inference engines on mainstream LLMs.

---

<sup>†</sup>These authors contributed equally to this work.

<sup>‡</sup>Prof. Guohao Dai is the Chief Scientist at Infinigence-AI, Ke Hong, Jiaming Xu, Qiuli Mao, and Jun Liu are interns at Infinigence-AI.

<sup>✉</sup>Prof. Guohao Dai and Prof. Yu Wang are the corresponding authors of this paper.

## 1 Introduction

As the Large Language Model (LLM) achieved unprecedented success in various domains [2, 3, 4, 5], the LLM inference workload is skyrocketing. For example, OpenAI reports that GPT-4 inference with 8K context length costs \$0.03 per 1K input tokens and \$0.06 per 1K output tokens [6]. Currently, OpenAI has 180.5 million users and receives over 10 million queries per day [7]. Consequently, the cost to operate OpenAI’s model like ChatGPT is approximately \$7 million per day for the necessary computing hardware [8]. Thus, optimizations on LLM inference performance will have a huge impact considering massive LLM inference scenarios. Many recent works have proposed techniques to accelerate LLM inference tasks, including DeepSpeed [9], FlexGen [10], vLLM [11], OpenPPL [12], FlashDecoding [13], TensorRT-LLM [14], and etc [15, 16, 17, 12].

The LLM inference task generates tokens (*e.g.*, words) from the input sequence autoregressively, and can be organized into two typical phases: the *prefill* phase and the *decode* phase. The *prefill* phase generates the first token by processing the input prompt, and previous research (*e.g.*, FlashAttention [18, 19]) optimizes latency for this phase. The *decode* phase generates the following tokens sequentially, and many works [9, 10, 11, 15, 13, 14, 20] focus on improving the throughput of generating tokens (*i.e.*, reducing latency of each token). The *prefill* phase dominates total time for scenarios of long-sequence input or generating short outputs [21, 22], while the *decode* phase constitutes a significant portion of the time when processing long output sequences [23].

Figure 2 shows the main dataflow of the LLM inference with one transformer layer for both the *prefill* phase and the *decode* phase. A transformer layer can be divided into linear GEMM (General Matrix Multiplication) operations (*e.g.*,  $K$ ,  $Q$ ,  $V$  weight projection and the feedforward) and the attention/softmax computation. For the attention computation, a softmax operation is adopted for a row in the attention matrix. To improve the parallelism, previous designs [18, 13] divide the attention matrices into smaller tiles and rows are also split to compute partial softmax results. A synchronized softmax operation is adopted to update previous partial softmax results when a new partial softmax result is calculated. Such a **synchronized partial softmax update** accounts for 18.8% for the attention computation of Llama2-7B inference according to our profiling on NVIDIA Tesla A100 GPU with 1024 input length, resulting in the first challenge for accelerating LLM inference. Secondly, **the computation resources is under-utilized for the flat GEMM operation** during the *decode* phase. Because the *decode* phase sequentially generates tokens, the linear GEMM operation tends to be flat-shape (even turning into the GEMV (General Matrix-Vector Multiplication) operation when the batch size is 1). For the small batch size (*e.g.*, 8), previous designs [24, 25] pad the matrix with zeros to perform GEMMs of larger sizes (*e.g.*, 64), leading to over 50% computation under-utilization. Thirdly, **the performance of LLM inference suffers from the static dataflow** considering input dynamics and hardware configuration. For example, the small batch size makes the *decode* phase of LLM inference memory-bounded and the large batch size makes it compute-bounded. A single and static dataflow may lead to 50.25% performance loss for GEMMs of different shapes in LLM inference.

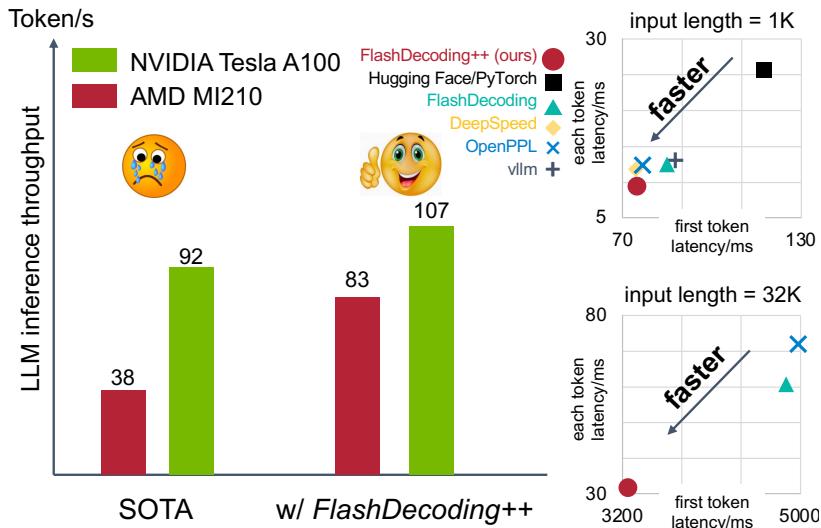


Figure 1: Overview of comparison between *FlashDecoding++* and state-of-the-art designs. The results in the figure are reported with Llama2-7B model [1]. The left is with batch size=1 and input length=1K, and TensorRT-LLM and Hugging Face are the SOTA baseline for NVIDIA/AMD according to our experimental results. The right shows the comprehensive comparison of both first token latency and each token latency.

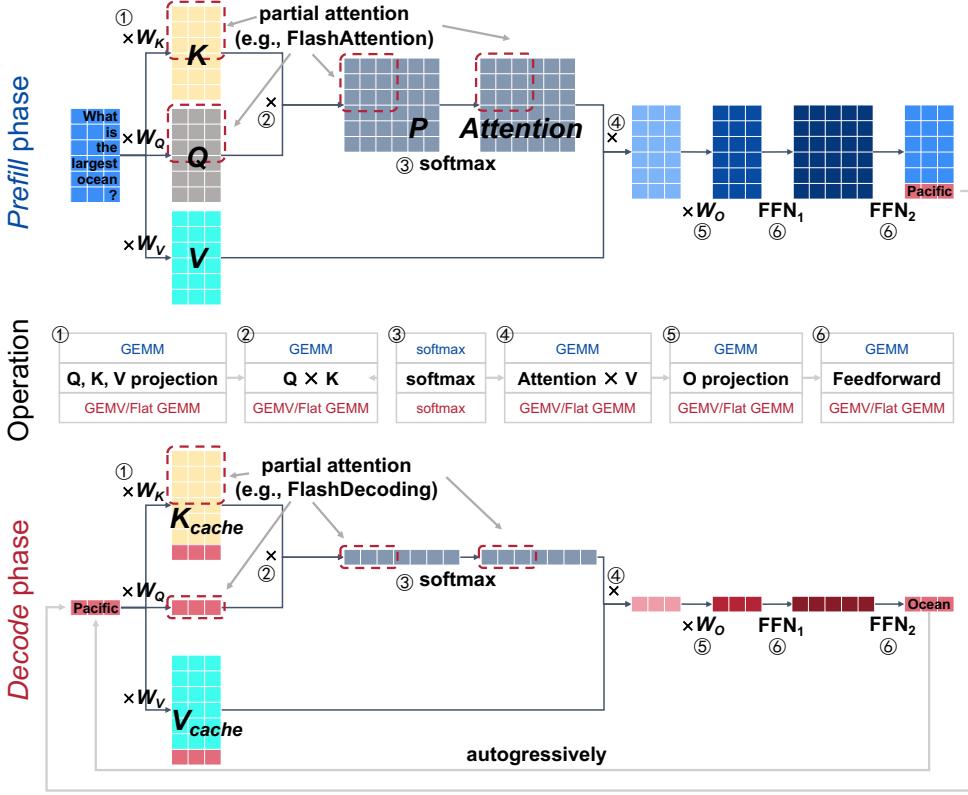


Figure 2: Overview of Large Language Model inference dataflow. We show the dataflow comparison between the *prefill* phase and the *decode* phase. The *prefill* phase mainly involves the GEMM operation, while the *decode* phase mainly involves the GEMV/Flat GEMM operation.

To tackle these challenges and enable a faster Large Language Model (LLM) inference, we present *FlashDecoding++* in this paper. *FlashDecoding++* creatively proposes the following contributions:

- **Asynchronized softmax with unified max value.** *FlashDecoding++* leverages a unified max value for different partial softmax computations. Each partial softmax result can be processed individually without synchronized update.
- **Flat GEMM optimization with double buffering.** *FlashDecoding++* only pads the matrix size to 8 rather than 64 in previous designs for flat-shaped GEMM to improve computation utilization. We point out that flat GEMMs with different shapes face varied bottlenecks, and further improve the kernel performance with techniques like double buffering.
- **Heuristic dataflow with hardware resource adaption.** *FlashDecoding++* takes both input dynamics and hardware configurations into consideration and dynamically applies kernel optimization for the LLM inference dataflow.

Because of the versatility of optimizations, the effectiveness of *FlashDecoding++* can be proved on both NVIDIA and AMD GPUs. *FlashDecoding++* achieves up to **4.86×** and **2.18×** speedup on both NVIDIA and AMD GPUs compared with Hugging Face implementations, respectively. Our extensive results show that *FlashDecoding++* achieves an average of **1.37×** speedup compared with FlashDecoding [13], a state-of-the-art LLM inference engine on various LLMs (e.g., Llama2, ChatGLM2, etc.).

The rest of this paper is organized as follows. Section 2 introduces preliminaries of LLMs and related works on LLM inference acceleration. Our three techniques, the asynchronized softmax with unified max value, the flat GEMM optimization with double buffering, and the heuristic dataflow with hardware resource adaption are detailed in Section 3, 4, and 5, respectively. Section 6 presents the evaluation results. Related works on LLM inference are introduced in Section 7, and Section 8 concludes the paper.

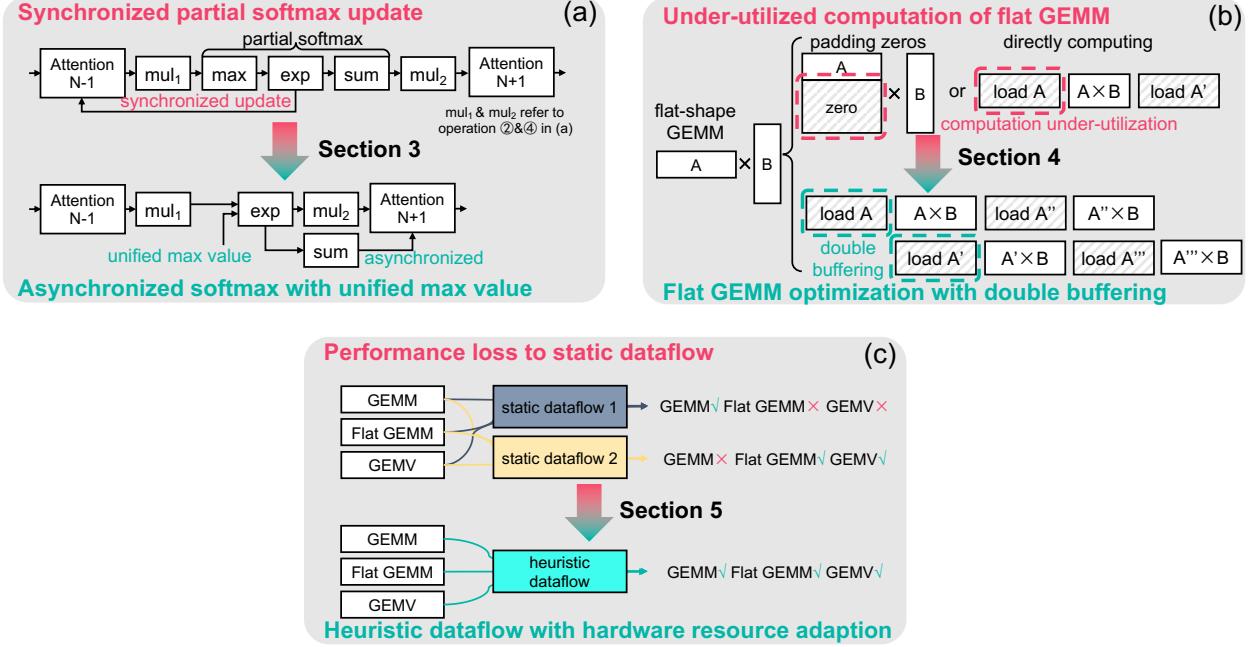


Figure 3: *FlashDecoding++* proposes three solutions for corresponding challenges in Large Language Model inference. (a) *FlashDecoding++* proposes the asynchronous softmax with unified max value technique, avoiding synchronized update to previous partial attention results. (b) *FlashDecoding++* optimizes flat GEMM by improving computation utilization. (c) *FlashDecoding++* heuristically optimizes dataflow.

## 2 Background

### 2.1 LLM Inference Dataflow Overview

The task of LLM inference is to generate tokens from the input sequence, which can be used to complete a sentence or answer a question. An overview of the LLM inference dataflow is shown in Figure 2. As we can see, the LLM inference dataflow can be organized into two typical phases with similar operations: one *prefill* phase and several *decode* phases. The *prefill* phase “understands” the input sequence (*i.e.*, “What is the largest ocean?”). Each token (we set one word as a token in Figure 2) is encoded as an embedding vector, and the input sequence is organized into a matrix. The main output of the *prefill* phase is a new token, which is predicted to be the next token after the input sequence (*i.e.*, “Pacific” in this figure). The *decode* phase “generates” the output sequence (*i.e.*, “Pacific”, “Ocean”, etc.) The output token of the *prefill* phase is taken as the input of the *decode* phase. The *decode* phase is executed autogradively, and each output token is used as the input token for the next. The *decode* (*e.g.*, “Ocean” is further used as the input).

### 2.2 Operations in LLM Inference

The main operations in LLM inference are depicted as operation ① to ⑥ in Figure 2, including the linear projection (① and ⑤), the attention (②, ③, and ④), and the feedforward network (⑥). For simplicity, operations like position embedding [26], non-linear activation [27, 28, 29], mask [26], and others are not shown in the figure. Operations in the *prefill* phase and the *decode* phase are different in the shape of data. Because only one token (batch size=1) or few tokens (batch size>1) are processed at one time, **input matrices in the *decode* phase are flat-shape matrices or even vectors.**

**Linear Projection.** The linear projection performs as the fully connected layer, multiplying the input with weight matrices (*i.e.*,  $W_K, W_Q, W_V, W_O$ , called  $K, Q, V$  projection and  $O$  projection). For the *prefill* phase, the  $K, Q, V$  projection generates matrices  $K, Q, V$ . For the *decode* phase, the  $K, Q, V$  projection generates three corresponding vectors and concatenated with  $K$  and  $V$  (*i.e.*, KVcache, yellow and light blue in Figure 2 in the *prefill* phase).

$$\text{softmax}(Q \times K^T) \times V \quad (1)$$

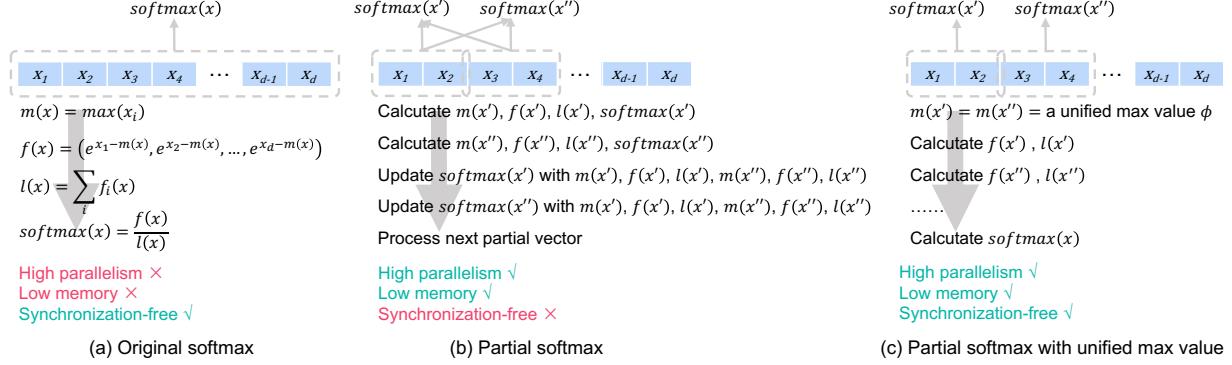


Figure 4: Comparison of different softmax computation schemes. (a) Softmax computation for the whole vector. (b) Computing partial softmax for each partial vector, and a synchronized update operation is required for all partial softmax results. (c) Computing partial softmax using a unified max value, and each partial vector is processed individually without synchronized update.

**Attention.** The attention operation is mainly divided into three operations (② to ④  $Q \times K$ ,  $\text{softmax}$ ,  $\text{Attention} \times V$ ), as shown in Eq. (1). For  $P = Q \times K^T$ , the softmax operation is performed for each row of the result matrix of  $P$ . The detailed softmax computation is shown in Figure 4(a). The maximum value  $m(x)$  is first calculated. The exponent of each element divided by  $e^{m(x)}$ ,  $f(x)$ , is then processed. These exponents are normalized to the summation of all exponents (*i.e.*,  $l(x)$ ) to get the softmax result.

**Feedforward Network.** The feedforward network primarily comprises two fully connected layers. The first one (⑥  $FFN_1$ ) expands the feature dimensions to enhance the representational capacity. The second one (⑥  $FFN_2$ ) restores the feature dimensions and serves as the output layer.

### 2.3 Attention Optimization

The softmax operation shown in Figure 4(a) requires all global data to be calculated and stored before it can proceed. This results in high memory consumption and low parallelism. Latter works propose the partial softmax technique to reduce memory consumption [18, 19] or improve parallelism [13]. Figure 4(b) shows the diagram of the partial softmax operation. The main idea is to divide the vector  $x$  into partial vectors (*i.e.*,  $x'$  and  $x''$ ). The partial softmax results of  $x'$  and  $x''$  are calculated separately according to Figure 4(a), and then synchronously updated by each other. The detailed computation of this synchronized update is shown in Equation (2). With the implementation of partial softmax, we can achieve efficient parallelism of computation while reducing memory cost for attention computation.

$$\begin{aligned}
 m(x) &= \max(m(x'), m(x'')) \\
 f(x') &= e^{m(x') - m(x)} f(x') \\
 f(x'') &= e^{m(x'') - m(x)} f(x'') \\
 l(x) &= f(x') + f(x'') \\
 \text{softmax}([x', x'']) &= [f(x'), f(x'')] \div l(x)
 \end{aligned} \tag{2}$$

However, since the partial softmax needs to be updated according to other partial softmax results, it unavoidably introduces data synchronization operations. According to our profiling result, such a synchronized update operation leads to 18.8% overheads in the attention computation for Llama2-7B inference on NVIDIA Tesla A100 GPU with 1024 input length.

## 3 Asynchronized Softmax with Unified Maximum Value

**Motivation.** The partial softmax operation requires synchronization among different partial vectors, leading to ~20% overheads of the attention operation. As is shown in Figure 2, the synchronization is required after the maximum value of the partial vector is calculated. The maximum value is used to update previous partial softmax (*i.e.*, recompute previous attention) results. Thus, to reduce synchronization overheads, **the key problem to be solved is how to compute each partial softmax result without requiring results from other partial softmax computation.**

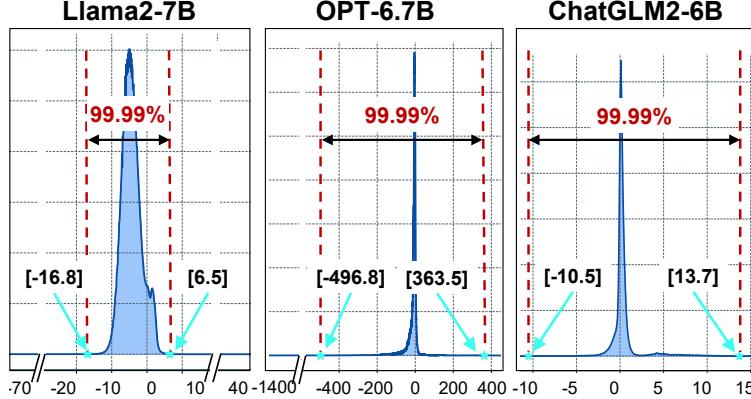


Figure 5: The statistical distribution of  $x_i$  (elements in the input vectors of softmax) in typical LLMs with different inputs.

**Challenge.** The reason that synchronization is required lies in that the maximum value of each partial vector is different. The maximum value is used to avoid overflow of the exponent operation ( $f(x)$  in Figure 4(a)), and exponents are summed ( $l(x)$  in Figure 4(a)) as the denominator of the softmax operation. Such a non-linear operation on each partial maximum value makes the synchronization among each partial softmax computation unavoidable.

**Analysis and Insights.** According to the formula of softmax computation, the maximum value is used as the scaling factor for both the numerator and the denominator (*i.e.*,  $f(x)$  and  $l(x)$  in Figure 4(a)). Our key insight is, **the scaling factor can be an arbitrary number** rather than using the maximum value mathematically, shown in Equation (3). When we set  $\phi = 0$ , it becomes the original softmax computation [30].

$$\begin{aligned} \text{softmax}(x) &= \frac{[e^{x_1-m(x)}, \dots, e^{x_d-m(x)}]}{\sum_i e^{x_i-m(x)}} \\ &= \frac{[e^{x_1-\phi}, \dots, e^{x_d-\phi}]}{\sum_i e^{x_i-\phi}}, \forall \phi \in \mathbb{R} \end{aligned} \quad (3)$$

However, the scaling factor cannot be an arbitrary number considering the overflowing of the exponent computation. For the case where  $x_i \gg \phi$ ,  $e^{x_i-\phi}$  overflows and cannot be represented using a fix-width floating point number (*e.g.*, float32 for exponent results in current LLM engines). For another case where  $x_i \ll \phi$ ,  $e^{x_i-\phi} \rightarrow 0$ , leading to precision loss. Thus, a proper scaling factor  $\phi$  should be carefully selected to avoid the two cases above. Figure 5 shows the statistical distribution of  $x_i$  (elements in the input vectors of softmax) in typical LLMs with different inputs [31]. Our key insight is, **> 99.99%  $x_i$  are within a certain range**. Specifically, for Llama2-7B, we have  $-16.8 < x_i < 6.5$  for **> 99.99%  $x_i$** . Because  $e^{b-a}$  and  $e^{a-b}$  can be represented by a float32 format, we can set  $\phi = a$  in Equation (3). For OPT-6.7B, we do not apply the technique in this section because of the large range in Figure 5.

**Approach: Asynchronization.** Based on the insights above, each partial softmax computation shares a unified maximum value,  $\phi$ . After the softmax operation, an inner product operation is executed between the softmax result and a column of  $V$  (*i.e.*,  $v$ ). Assume that the input vector  $x$  can be divided into  $p$  partial vectors,  $x = [x^{(1)}, \dots, x^{(p)}]$  ( $v = [v^{(1)}, \dots, v^{(p)}]$  correspondingly), we have:

$$\begin{aligned} \langle \text{softmax}(x), v \rangle &= \frac{\sum_i e^{x_i-\phi} \cdot v_i}{\sum_i e^{x_i-\phi}} \\ &= \frac{\sum_{j=1}^p \sum_{i=1}^{d/p} e^{x_i^{(j)}-\phi} \cdot v_i^{(j)}}{\sum_{j=1}^p \sum_{i=1}^{d/p} e^{x_i^{(j)}-\phi}} \end{aligned} \quad (4)$$

The inner accumulation in both the numerator and the denominator only take the partial vectors  $x^{(j)}$  and  $v^{(j)}$  as input, thus they can be processed asynchronously and individually. The outer accumulation is only processed after all partial vectors are processed. As we can see in Figure 4(c), each  $f(x^{(j)})$  is calculated individually, and  $\text{softmax}(x)$  is calculated after all  $x^{(j)}$  is calculated.

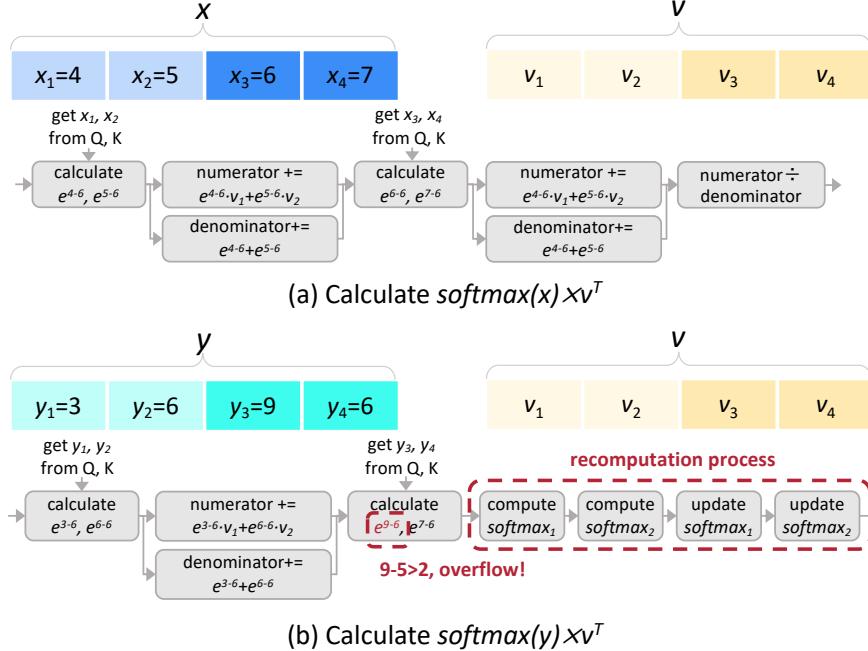


Figure 6: Example of asynchronous partial softmax computation. (a) Each partial softmax result is process individually without the synchronized update. (b) The recomputation process for all parital softmax computation is required when overflow happens.

**Approach: Recomputation.** Without loss of generality, we assume  $a < x_i - \phi < b$  for each  $x_i$  to ensure precision and avoid overflow. Then, the partial softmax operation is processed individually. However, when  $x_i - \phi \leq a$  or  $x_i - \phi \geq b$ , the asynchronous partial softmax computation is terminated for the vector  $x$  where  $x_i$  belongs to. The softmax is then recomputed using the synchronized partial softmax scheme (used in FlashAttention [18, 19] and FlashDecoding [13]) shown in Figure 4(b). Such a recomputation scheme avoids overflow while introducing negligible overheads based on the statistical data shown in Figure 5.

**Example.** Figure 6 shows an example of the asynchronous softmax scheme. We set  $a = -3, b = 3, \phi = 6$ . Two vectors  $x$  and  $y$  are calculated from  $Q \times K^T$  in Equation (1), and are divided into 2 partial vectors. We omit the process from  $Q \times K^T$  to these partial vectors. For each  $x_i$ , we have  $a < x_i - \phi < b$ , we process  $e^{x_1-\phi} \cdot v_1 + e^{x_2-\phi} \cdot v_2$  and  $e^{x_1-\phi} + e^{x_2-\phi}$  for the first partial vector of  $x$  using two asynchronous threads. Then, each thread moves to the next partial vector for the corresponding computation (*i.e.*,  $e^{x_3-\phi} \cdot v_3 + e^{x_4-\phi} \cdot v_4$  and  $e^{x_3-\phi} + e^{x_4-\phi}$ ). Two threads are synchronized when all partial vectors are processed, and perform the division operation in Equation (4). For  $y$ , the first partial vector is processed similarly. However, we find that  $y_3 - \phi > b$ , then two threads are terminated and the first thread recomputes all partial vectors according to the synchronized partial softmax scheme in Figure 4(b).

## 4 Flat GEMM Optimization with Double Buffering

**Motivation.** The process of the *decode* phase is mainly composed of GEMV (batch size=1) or flat GEMM (batch size>1) operation. Without loss of generality, GEMV/GEMM operations can be represented using  $M, N, K$ , where the sizes of two multiplied matrices are  $M \times K$  and  $K \times N$ . Previous LLM inference engines utilize Tensor Core to accelerate these operations using libraries like cuBLAS [24] and CUTLASS [25]. Although modern Tensor Core architectures [32] process GEMM with  $M = 8$ , these libraries usually tile the  $M$ -dimension to 64 to hide memory latency. However, for GEMV or flat GEMM operations in the *decode* phase, we usually have  $M \ll 64$  and the  $M$ -dimension is padded to 64 with zeros. The padding leads to under-utilized computation, and **the key problem is to process GEMV or flat GEMM operations with smaller tiles (*i.e.*, padding to 8 corresponding to modern Tensor Core architectures) in the  $M$ -dimension.**

**Challenge.** Processing GEMV or flat GEMM operations is non-trivial when the  $M$ -dimension is padded to 8. The tiling technique in modern libraries like cuBLAS [24] and CUTLASS [25] can only be applied to the  $N$ -dimension and

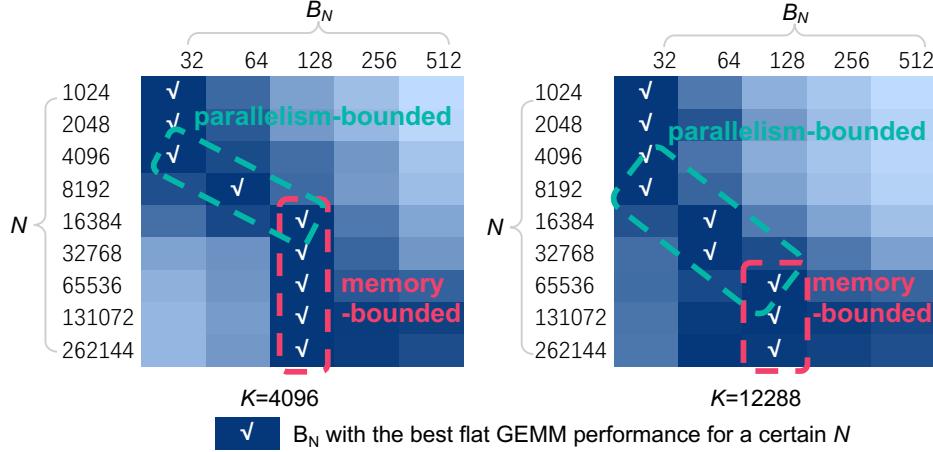


Figure 7: Normalized flat GEMM performance under different  $N$ -dimension sizes and  $N$ -dimension tiling sizes. We set  $M = 8$  and execute GEMM on the NVIDIA Tesla A100 GPU.

the  $K$ -dimension. Tiles on the  $K$ -dimension are processed sequentially in a GPU block to avoid atomic operations during reduction. Tiling on the  $N$ -dimension affects both parallelism and computation/memory ratio, which are both important for GEMV and flat GEMM acceleration.

**Analysis and Insights.** Assume that tiling sizes of the  $N$ -dimension and the  $K$ -dimension are  $B_N$  and  $B_K$ , respectively. The computation of each GEMM tile is  $2 \times M \times B_N \times B_K$  with total  $B = \frac{N \times K}{B_N \times B_K}$  GEMM tiles. The total memory access is  $(M \times B_K + B_N \times B_K) \times B + M \times N$ . Thus, the computation/memory ratio is:

$$\begin{aligned} & \frac{2 \times M \times B_N \times B_K \times B}{(M \times B_K + B_N \times B_K) \times B + M \times N} \\ &= \frac{2 \times M \times K}{K + \frac{M \times K}{B_N} + M} \end{aligned} \tag{5}$$

On the other hand, the parallelism is  $\frac{N}{B_N}$ . Thus, the computation/memory ratio shows a positive correlation with  $B_N$  while the parallelism shows a negative correlation with  $B_N$ , exposing a contradiction on improving the performance of GEMV or flat GEMM. We depict the normalized performance of the flat GEMM in Figure 7 with different  $N$  and  $B_N$ . Our key insight is, **for the smaller  $N$ , the flat GEMM is parallelism-bounded**. There are 108 Streaming Multiprocessors (SMs) in the NVIDIA Tesla A100.  $\frac{N}{B_N}$  tends to be a constant (e.g., 128 or 256), which is related to the hardware parallelism (number of SMs). Another key insight is, **for the larger  $N$ , the flat GEMM becomes memory-bounded**. The performance of these cases can be improved by hiding memory access latency.

**Approach: Double Buffering.** In order to hide memory access latency, we introduce the double buffering technique for the flat GEMM operation. We allocate two separate buffers in the shared memory. The tile in one buffer performs the GEMM operation, while another buffer loads a new tile for the next GEMM operation. Thus, the computation and the memory access are overlapped. We apply such a technique when  $N$  is large in our practice.

**Example.** Figure 8 shows the example of our flat GEMM optimization with double buffering. For  $M < 8$ , the  $M$ -dimension is first padded to 8 considering modern Tensor Core architectures. Workloads in the  $K$ -dimension are processed within one GPU block (e.g.,  $A_1, A_2, A_3, \dots$ ), while workloads in the  $N$ -dimension are processed in parallel using different GPU blocks (e.g.,  $C_1, C_2, \dots$ ). We take GPU Block<sub>1</sub> as an example, the first tile for each matrix in the  $K$ -dimension (i.e.,  $A_1$  and  $B_1$ ) is loaded to the left buffer in the shared memory. Then, the GEMM operation is performed between  $A_1$  and  $B_1$ . Consequently,  $A_2$  and  $B_2$  are loaded to the right buffer in the shared memory. The following tiles are processed similarly according to the double buffering scheme.

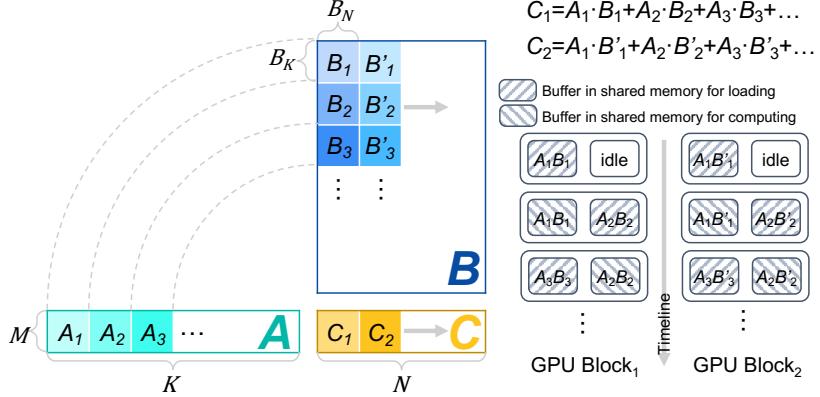


Figure 8: Double buffering for flat GEMM when  $N$ -dimension is large. The  $M$ -dimension is padded to 8 and not tiled.

## 5 Heuristic Dataflow with Hardware Resource Adaption

**Motivation.** Although *FlashDecoding++* optimizes the flat GEMM operation in Section 4, it does not cover all operations (even only for GEMMs) in the LLM inference. As mentioned in Figure 2, the shapes of GEMMs in different operations and two phases vary. Thus, the GEMM workload in the LLM inference can be GEMV (batch size=1 for the *decode* phase), flat GEMM (small batch size for the *decode* phase and short sequence length for the *prefill* phase) and conventional GEMM (large batch size or long sequence length for the *prefill* phase). In order to leverage the powerful computational ability of Tensor Core, current frameworks like FasterTransformer [33] and DeepSpeed [9] tend to utilize the highly optimized GEMM implementation from cuBLAS [24] to deal with different workloads. However, the Tensor Core implementation fails with the GEMV workload. The GEMV workload can be optimized by utilizing CUDA Core in previous designs like FastGEMV [34]. For a Llama2-7B linear layer in the *decode* phase, the Tensor Core implementation from cuBLAS only achieves 82.15% of the performance of CUDA Core implementation using FastGEMV on an NVIDIA A100 GPU. On the other hand, using CUDA Core to do the projection on a batchsize=4 decoding input only achieves 49.75% performance compared with the Tensor Core implementation. Thus, in order to approach the optimal computation performance, **a heuristic dataflow is supposed to be exploited in for different workloads.**

**Challenge.** Although a heuristic dataflow potentially exists in the implementation of different linear workloads, it is challenging to build the mapping from a certain workload to an optimal implementation. In the scenario of LLM inference, there are various factors that influence the implementation performance of linear workloads: (a) Input dynamics. The variety of the batch size and the input sequence length brings dynamic workloads. (b) Model diversity. The linear workload varies with different model structures and sizes. (c) GPU capacities. The relative performance between implementations changes with GPU characteristics, such as memory bandwidth, cache size, and computational ability. (d) Engineering effects. The engineering effort also highly impacts the kernel performance. All these influential factors build a large search space, making it non-trivial to generate an effective mapping between the linear workload and the corresponding optimal implementation.

**Analysis and Insights.** Although all influential factors form a large search space, the homogeneity of different layers in LLM significantly reduces the search space for operator optimization. Figure 2 shows four linear GEMV/GEMM operations in the *prefill* phase and the *decode* phase, *i.e.*,  $K, Q, V$  projection,  $O$  projection, and two feedforward operations. Each GEMV/GEMM operation can be abstracted as a multiplication between an  $(M \times K)$ -shaped matrix and a  $(K \times N)$ -shaped matrix. Our key insight is, **there are only four  $[K, N]$  shapes for a certain LLM**. Moreover,  $M$  is only related to the input sequence length and the batch size for the *prefill* phase, and the batch size for the *decode* phase. Figure 9(a) shows limited shapes of GEMV/GEMM operations in the LLM inference.

**Approach: Decision flow for inflection points.** Because only four  $[K, N]$  shapes exist for a certain LLM, we use three types of implementations for GEMV/GEMM operations when  $M$  varies: FastGEMV for the GEMV and flat GEMM operations (ImplA), our flat GEMM optimization in Section 4 (ImplB), and the CUTLASS [25] libraries optimized for the conventional GEMM (ImplC). Thus, it is important to decide whether applying ImplA or ImplB for a small  $M$ , and ImplB or ImplC for a large  $M$ . Figure 9(b) shows the decision flow. *FlashDecoding++* profiles the performance of ImplA and ImplB for a certain  $M$ , and increases  $M$  to find an inflection point  $M_1$  where the performance of ImplB is

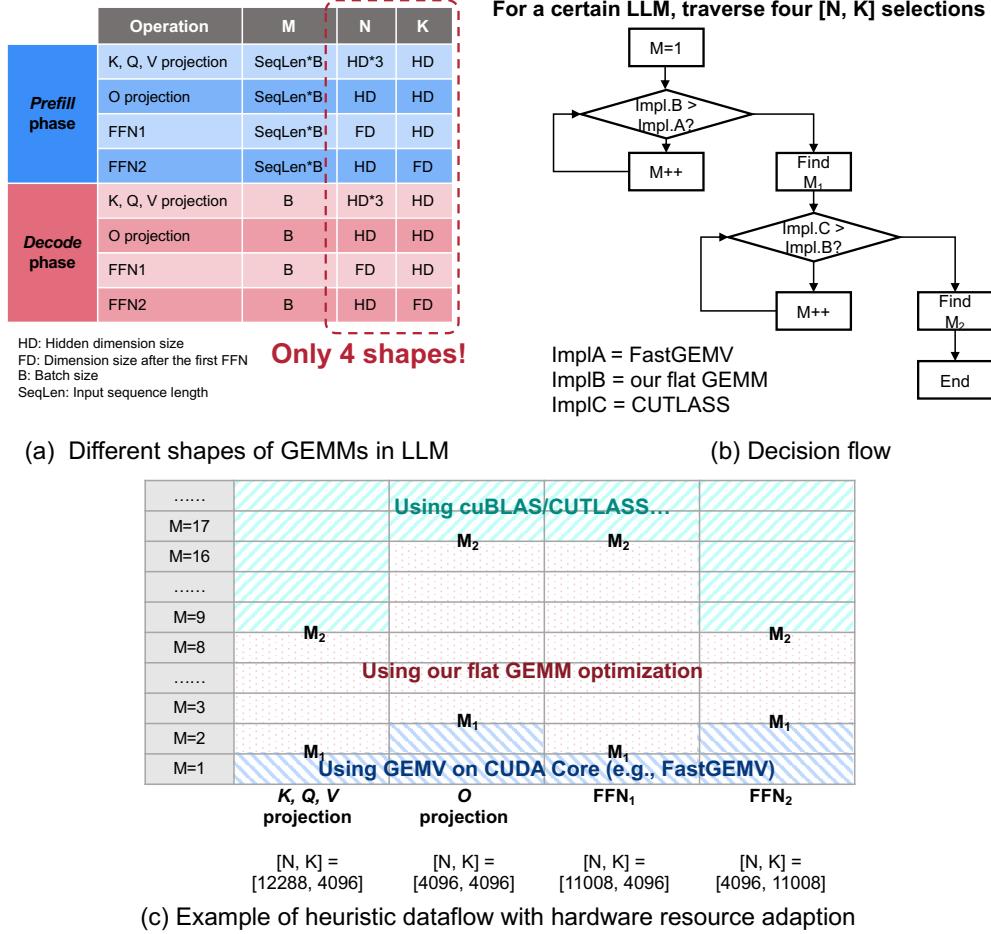


Figure 9: Heuristic dataflow with hardware resource adaption in *FlashDecoding++*. (a) Only four  $[N, K]$  shapes exist for a certain LLM. (b) The decision flow. We traverse all  $[N, K]$  selections and profile the performance of three representative implementations.  $M$  is increased to find two inflection points for runtime heuristic dataflow. (c) *FlashDecoding++* heuristically utilizes Tensor Core/CUDA Core with the corresponding GEMV/GEMM implementation by referring to a lookup table.

better than  $\text{ImplA}$ . Another inflection point  $M_2$  is found similarly where the performance of  $\text{ImplC}$  is better than  $\text{ImplB}$ . Note that each  $[N, K]$  gets its individual  $M_1$  and  $M_2$ .

**Approach: Heuristic dataflow.** For the runtime LLM inference, *FlashDecoding++* adopts  $\text{ImplA}$  using CUDA Core when  $M < M_1$ , and  $\text{ImplB}/\text{ImplC}$  using Tensor Core when  $M_1 \leq M < M_2/M_2 \leq M$ . Note that the decision flow are executed offline, it does not affect the performance of runtime LLM inference.

**Example.** Figure 9(c) shows an example of applying the heuristic dataflow for the Llama2-7B model. Four  $[N, K]$  shapes are [12288, 4096] for  $K, Q, V$  projection, [4096, 4096] for  $O$  projection, [11008, 4096] and [4096, 11008] for FFN. For each  $[N, K]$ , the inflection points are found based on the decision flow in Figure 9(c). Then, a lookup table is formed, and each GEMV/GEMM operation is executed according to corresponding implementations during runtime. In this example, FastGEMV is adopted for the  $K, Q, V$  projection when batch size=1 ( $M = 1$ ) for the *decode* phase, and our flat GEMM optimization is applied when batch size=1/input sequence length=8 for FFN<sub>1</sub> ( $M = 8$ ).

## 6 Evaluation

### 6.1 Experiments Setup

We evaluate the performance of *FlashDecoding++* on different GPUs with various Large Language Models. We compare the performance with several state-of-the-art LLM inference engines.

Table 1: Hardware Platforms

	NVIDIA		AMD	
GPU	Tesla A100 80 GB CUDA 12.2	RTX3090 24 GB CUDA 11.6	MI210 64GB ROCm 5.7	RX7900XTX 24GB ROCm 5.6
CPU	Intel Xeon Silver 8358P 2.60 GHz	Intel Xeon Gold 6226R 2.90GHz	AMD EPYC 7K62 2.60GHz	Intel Core i9-10940X 3.30GHz

Table 2: Model Configuration

Model	Dimension	Heads	Layers	Context Length
Llama2-7B	4096	32	32	4k
Llama2-13B	5120	40	40	4k
OPT-6.7B	4096	32	32	2k
ChatGLM2-6B	4096	32	32	32k

### 6.1.1 Hardware Platforms

We evaluate the performance of *FlashDecoding++* and other LLM engines on both NVIDIA and AMD platforms to make a comprehensive comparison. We choose two different GPUs for each platform: Tesla A100 and RTX3090 for NVIDIA, MI210 and RX7900XTX for AMD. We show the detailed configuration in Table 1.

### 6.1.2 LLM Engine Baselines

We implement our *FlashDecoding++* using the Pytorch-based front-end with the C++ and CUDA backend for NVIDIA GPUs while ROCm for AMD GPUs. We compare the inference performance in both *prefill* phase and *decode* phase with the following LLM engine baselines: Hugging Face (HF) [35], vLLM [11], DeepSpeed [9], TensorRT-LLM [14], OpenPPL [12], and FlashAttention2/FlashDecoding [19, 13]. These baselines are introduced in Section 7.

### 6.1.3 Models

We evaluate the performance of *FlashDecoding++* with other LLM inference engines on three typical Large Language Models: Llama2, OPT, and ChatGLM2. Table 2 shows the detailed configuration of these models. Note that there may be several models in one LLM (*e.g.*, Llama2-7B, Llama2-13B) with different configurations (*e.g.*, number of heads and layers).

- **Llama2** [1] is a mainstream open-source LLM set released by Meta in 2023. It is a collection of pretrained and fine-tuned generative text models ranging in scale from 7B to 70B parameters.
- **OPT** [36], is a suite of decoder-only pre-trained transformers ranging from 125M to 175B parameters released by Meta AI.
- **ChatGLM2** [37] is an open-source LLM supporting bilingual (Chinese-English) chat.

## 6.2 Comparison with State-of-the-art

We compare *FlashDecoding++* with state-of-the-art LLM inference engines in Figure 10 and Figure 11 on NVIDIA GPUs, Figure 12 and Figure 13 for AMD GPUs. For the *decode* phase, *FlashDecoding++* achieves up to **4.86×** speedup compared with Hugging Face implementations on three LLMs and two GPUs. The average speedup over vLLM, DeepSpeed, TensorRT-LLM, OpenPPL, and FlashDecoding is 1.25×, 1.48×, 1.12×, 1.34×, and 1.24× (**1.37×** on Tesla A100 compared with FlashDecoding), respectively. For the *prefill* phase, *FlashDecoding++* achieves up to 1.40× speedup compared with Hugging Face implementations. The average speedup over DeepSpeed, TensorRT-LLM, OpenPPL, FlashAttention2 and FlashDecoding is 1.05×, 1.06×, 1.08×, 1.09×, and 1.08×, respectively. We also show the *decode* results on two AMD GPUs. Currently, only the original Hugging Face implementation can be executed on AMD GPUs as the baseline. *FlashDecoding++* achieves up to 2.08× and **2.18×** compared with the baseline on RX7900XTX and MI210, respectively.

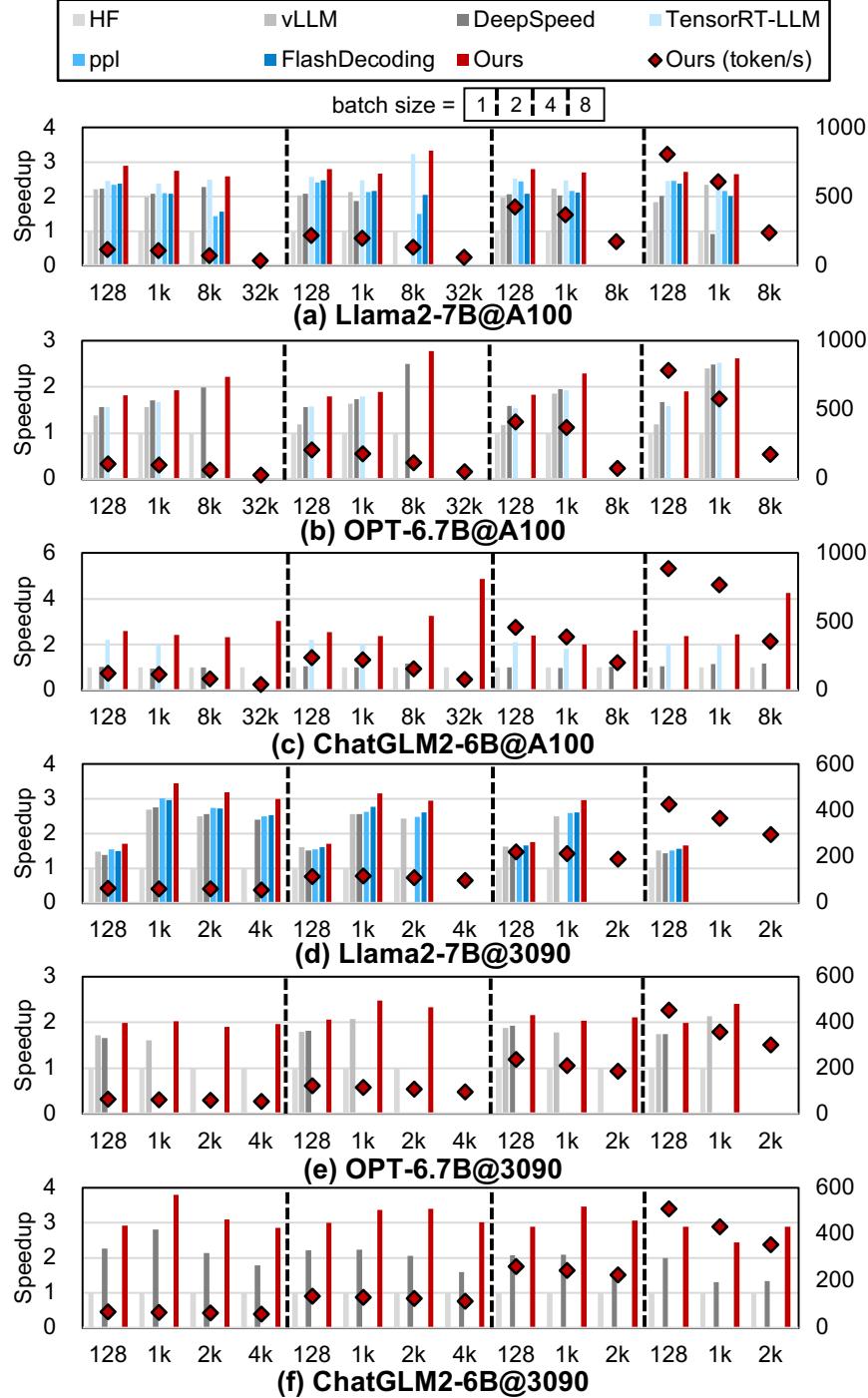
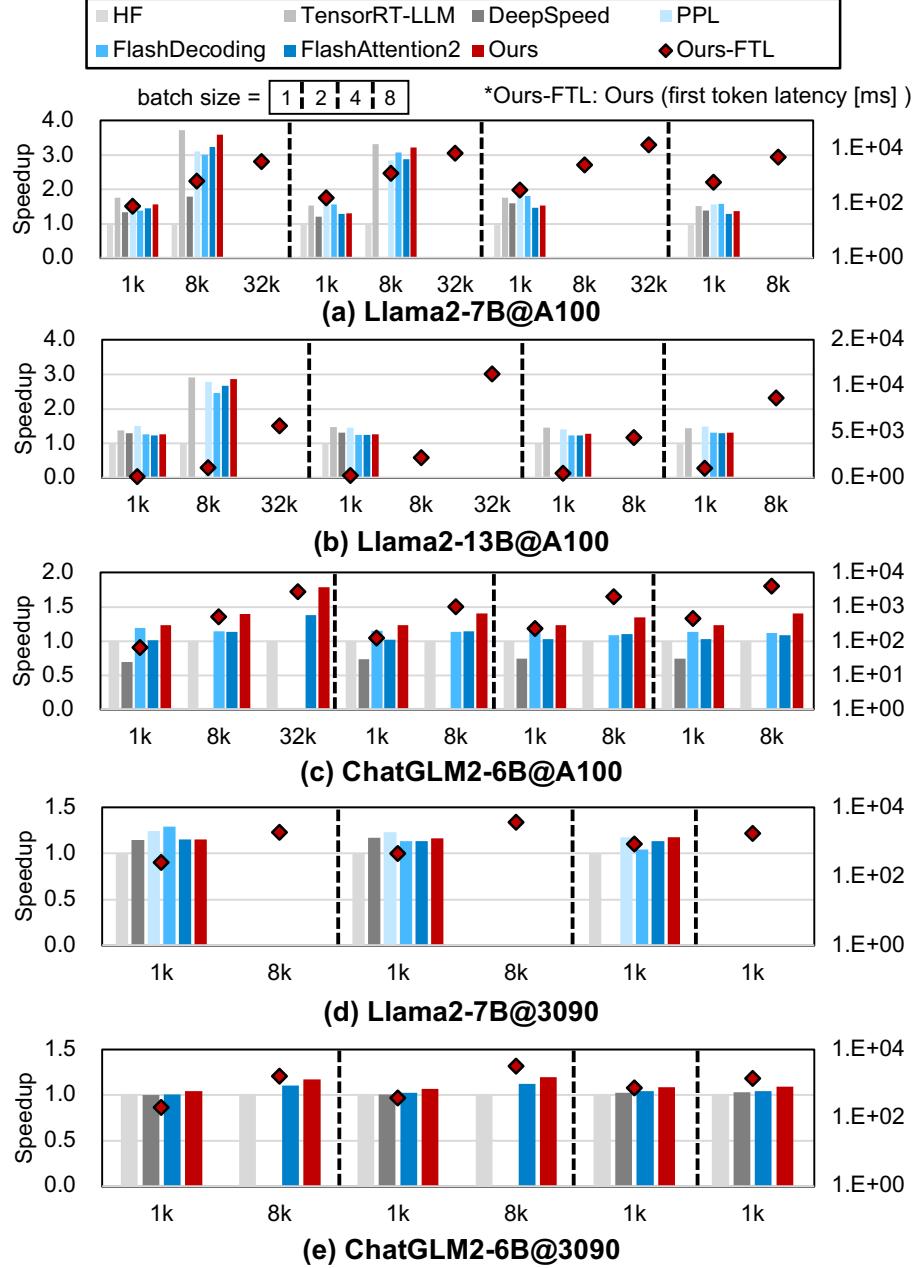


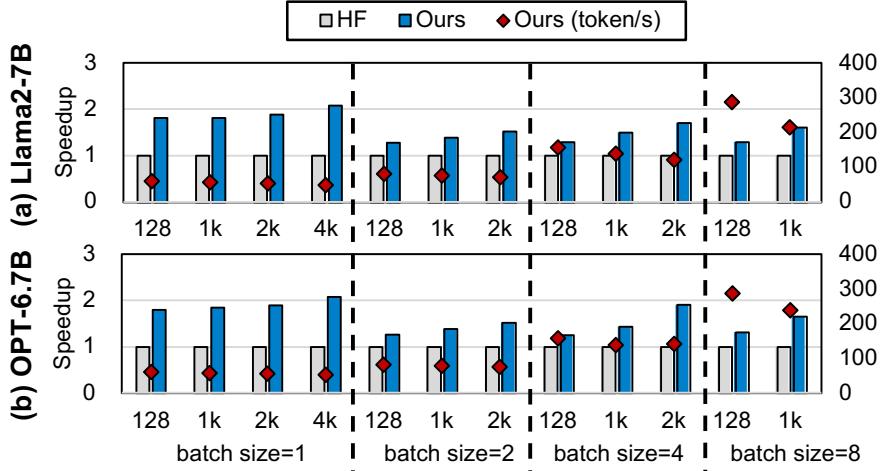
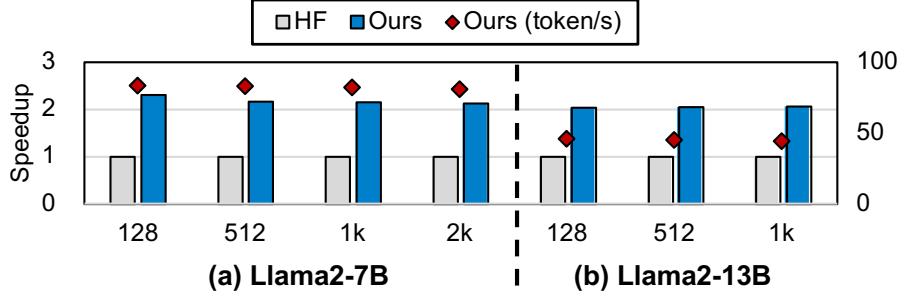
Figure 10: Speedup of the *decode* phase on NVIDIA GPUs. Blank bars represent the model cannot be executed (*e.g.*, OpenPPL does not support OPT-6.7B/ChatGLM2-6B, TensorRT-LLM fails to compile the model with > 8K input length, and etc.)

## 7 Related Works

Large language model inference acceleration has gained significant attention in recent research, with several notable approaches and techniques emerging in the field. **DeepSpeed** [9] is a comprehensive engine that optimizes both the

Figure 11: Speedup of the *prefill* phase on NVIDIA GPUs.

training and inference phases for LLMs. It achieves robust inference performance through kernel fusion and efficient GPU memory management, with a particular focus on optimizing memory usage for KVcache. **vLLM** [11] improves GPU memory utilization by efficient memory management techniques and the PageAttention method, leading to increased maximum batch sizes and elevating the upper limit of inference performance. **FlashAttention** [18, 19] optimizes the self-attention computation process during the prefill phase through improved parallelism and workload distribution. **FlashDecoding** [13] is an extension of FlashAttention and enhances the parallelism through splitting  $K$  and  $V$ , supporting efficient self-attention computation for long sequence during the decode phase. **FasterTransformer** [33] and **OpenPPL** [12] implement large model inference engines using C++ to reduce overhead resulting from kernels scheduling, compared to Python implementations. They also employ memory management techniques and kernel fusion to achieve efficient LLM inference. **TensorRT-LLM** [14] is built upon the *TensorRT* [38] and the *FasterTransformer*

Figure 12: Speedup of the *decode* phase on AMD RX7900XTX.Figure 13: Speedup of the *decode* phase on AMD MI210.

[33] engine (*C++*) and incorporates cutting-edge open-source technologies such as *FlashAttention* [18, 19]. Additionally, it enhances its ease of use by providing the *Python API*.

## 8 Conclusion

We propose *FlashDecoding++*, a fast Large Language Model inference engine in this paper. *FlashDecoding++* accelerates mainstream LLMs with multiple hardware backend support. *FlashDecoding++* proposes three novel designs: the asynchronous softmax with unified max value, the flat GEMM optimization with double buffering, and the heuristic dataflow with hardware resource adaption, achieving up to **4.86** $\times$  and **2.18** $\times$  speedup on NVIDIA and AMD GPUs compared with Hugging Face implementations. *FlashDecoding++* also achieves an average of **1.37** $\times$  speedup compared with state-of-the-art LLM inference engines, FlashDecoding, on various LLMs.

## References

- [1] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.
- [2] Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. Large language models in medicine. *Nature medicine*, 29(8):1930–1940, 2023.
- [3] Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay,

- Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernandez Abrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vlad Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, Guy Gur-Ari, Steven Hand, Hadi Hashemi, Le Hou, Joshua Howland, Andrea Hu, Jeffrey Hui, Jeremy Hurwitz, Michael Isard, Abe Ittycheriah, Matthew Jagielski, Wenhao Jia, Kathleen Kenealy, Maxim Krikun, Sneha Kudugunta, Chang Lan, Katherine Lee, Benjamin Lee, Eric Li, Music Li, Wei Li, YaGuang Li, Jian Li, Hyeontaek Lim, Hanzhao Lin, Zhongtao Liu, Frederick Liu, Marcello Maggioni, Aroma Mahendru, Joshua Maynez, Vedant Misra, Maysam Moussalem, Zachary Nado, John Nham, Eric Ni, Andrew Nystrom, Alicia Parrish, Marie Pellat, Martin Polacek, Alex Polozov, Reiner Pope, Siyuan Qiao, Emily Reif, Bryan Richter, Parker Riley, Alex Castro Ros, Aurko Roy, Brennan Saeta, Rajkumar Samuel, Renee Shelby, Ambrose Slone, Daniel Smilkov, David R. So, Daniel Sohn, Simon Tokumine, Dasha Valter, Vijay Vasudevan, Kiran Vodrahalli, Xuezhi Wang, Pidong Wang, Zirui Wang, Tao Wang, John Wieting, Yuhuai Wu, Kelvin Xu, Yunhan Xu, Linting Xue, Pengcheng Yin, Jiahui Yu, Qiao Zhang, Steven Zheng, Ce Zheng, Weikang Zhou, Denny Zhou, Slav Petrov, and Yonghui Wu. Palm 2 technical report, 2023.
- [4] Jan Clusmann, Fiona R Kolbinger, Hannah Sophie Muti, Zunamys I Carrero, Jan-Niklas Eckardt, Narmin Ghaffari Laleh, Chiara Maria Lavinia Löffler, Sophie-Caroline Schwarzkopf, Michaela Unger, Gregory P Veldhuizen, et al. The future landscape of large language models in medicine. *Communications Medicine*, 3(1):141, 2023.
  - [5] Can Cui, Yunsheng Ma, Xu Cao, Wenqian Ye, and Ziran Wang. Receive, reason, and react: Drive as you say with large language models in autonomous vehicles. *arXiv preprint arXiv:2310.08034*, 2023.
  - [6] OpenAI. Openai pricing. [Online], 2023. <https://openai.com/pricing>.
  - [7] Nerdynav. Up-to-date chatgpt statistics and user numbers [oct 2023]. [Online], 2023. <https://nerdynav.com/chatgpt-statistics>.
  - [8] AFZAL AHMAD DYLAN PATEL. The inference cost of search disruption - large language model cost analysis. [Online], 2023. <https://www.semianalysis.com/p/the-inference-cost-of-search-disruption>.
  - [9] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.
  - [10] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. 2023.
  - [11] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.
  - [12] Sensemte. Openppl: A high-performance deep learning inference platform. [Online], 2023. <https://openppl.ai/home>.
  - [13] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference. [Online], 2023. <https://crfm.stanford.edu/2023/10/12/flashdecoding.html>.
  - [14] Neal Vaidya, Fred Oh, and Nick Comly. Optimizing inference on large language models with nvidia tensorrt-llm, now publicly available. [Online], 2023. <https://github.com/NVIDIA/TensorRT-LLM>.
  - [15] Sensemte. A light and fast inference service for llm. [Online], 2023. <https://github.com/ModelTC/lightllm>.
  - [16] Text generation inference: Fast inference optimize for llms. [Online], 2023. <https://github.com/huggingface/text-generation-inference/>.
  - [17] Mlc llm: Machine learning compilation for large language models. [Online], 2023. <https://github.com/mlc-ai/mlc-llm>.
  - [18] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.
  - [19] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.
  - [20] Aaron Pham, Chaoyu Yang, Sean Sheng, Shenyang Zhao, Sauyon Lee, Bo Jiang, Fog Dong, Xipeng Guan, and Frost Ming. OpenLLM: Operating LLMs in production, June 2023.

- [21] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.
- [22] Z Dong, T Tang, L Li, and WX Zhao. A survey on long text modeling with transformers. arxiv 2023. *arXiv preprint arXiv:2302.14502*.
- [23] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.
- [24] NVIDIA. cublas: Basic linear algebra on nvidia gpus. [Online], 2017. <https://developer.nvidia.com/cublas>.
- [25] NVIDIA. Cutlass: Cuda templates for linear algebra subroutines. [Online], 2017. <https://github.com/NVIDIA/cutlass>.
- [26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [27] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.
- [28] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.
- [29] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.
- [30] John Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. *Advances in neural information processing systems*, 2, 1989.
- [31] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.
- [32] NVIDIA. Nvidia tensor core. [Online], 2023. <https://www.nvidia.com/en-us/data-center/tensor-cores/>.
- [33] NVIDIA. Fastertransformer: About transformer related optimization, including bert, gpt. [Online], 2017. <https://github.com/NVIDIA/FasterTransformer>.
- [34] Siping Wang. Fastgemv: High-speed gemv kernels. [Online], 2023. <https://github.com/wangsiping97/FastGEMV>.
- [35] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.
- [36] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuhui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.
- [37] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. Glm: General language model pretraining with autoregressive blank infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 320–335, 2022.
- [38] NVIDIA. Nvidia tensorrt: An sdk for high-performance deep learning inference. [Online]. <https://developer.nvidia.com/tensorrt>.

# FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU

Ying Sheng<sup>1</sup> Lianmin Zheng<sup>2</sup> Binhang Yuan<sup>3</sup> Zhuohan Li<sup>2</sup> Max Ryabinin<sup>4,5</sup> Daniel Y. Fu<sup>1</sup> Zhiqiang Xie<sup>1</sup>  
Beidi Chen<sup>6,7</sup> Clark Barrett<sup>1</sup> Joseph E. Gonzalez<sup>2</sup> Percy Liang<sup>1</sup> Christopher Ré<sup>1</sup> Ion Stoica<sup>2</sup> Ce Zhang<sup>3</sup>

## Abstract

The high computational and memory requirements of large language model (LLM) inference make it feasible only with multiple high-end accelerators. Motivated by the emerging demand for latency-insensitive tasks with batched processing, this paper initiates the study of high-throughput LLM inference using limited resources, such as a single commodity GPU. We present FlexGen, a high-throughput generation engine for running LLMs with limited GPU memory. FlexGen can be flexibly configured under various hardware resource constraints by aggregating memory and computation from the GPU, CPU, and disk. By solving a linear programming problem, it searches for efficient patterns to store and access tensors. FlexGen further compresses the weights and the attention cache to 4 bits with negligible accuracy loss. These techniques enable FlexGen to have a larger space of batch size choices and thus significantly increase maximum throughput. As a result, when running OPT-175B on a single 16GB GPU, FlexGen achieves significantly higher throughput compared to state-of-the-art offloading systems, reaching a generation throughput of 1 token/s for the first time with an effective batch size of 144. On the HELM benchmark, FlexGen can benchmark a 30B model with a 16GB GPU on 7 representative sub-scenarios in 21 hours. The code is available at <https://github.com/FMIInference/FlexGen>.

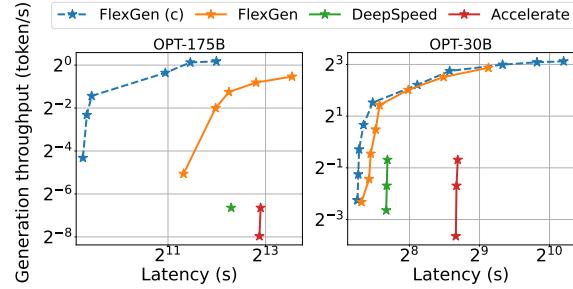


Figure 1. The total latency for a block and throughput trade-offs of three offloading-based systems for OPT-175B (left) and OPT-30B (right) on a single NVIDIA T4 (16 GB) GPU with 208 GB CPU DRAM and 1.5TB SSD. FlexGen achieves a new Pareto-optimal frontier with 100× higher maximum throughput for OPT-175B. Other systems cannot further increase throughput due to out-of-memory issues. “(c)” denotes compression.

## 1. Introduction

In recent years, large language models (LLMs) have demonstrated strong performance across a wide range of tasks (Brown et al., 2020; Bommasani et al., 2021; Zhang et al., 2022; Chowdhery et al., 2022). Along with these unprecedented capabilities, generative LLM inference comes with unique challenges. These models can have billions, if not trillions of parameters (Chowdhery et al., 2022; Fedus et al., 2022), which leads to extremely high computational and memory requirements to run. For example, GPT-175B requires 325GB of GPU memory simply to load its model weights. Fitting this model onto GPUs would require at least five A100 (80GB) GPUs and complex parallelism strategies (Pope et al., 2022; Aminabadi et al., 2022). Thus, lowering LLM inference resource requirements has recently attracted intense interest.

In this paper, we focus on a setting that we call *throughput-oriented generative inference*. In addition to interactive use cases such as chatbots, LLMs are also applied to many “back-of-house” tasks such as benchmarking (Liang et al., 2022), information extraction (Narayan et al., 2018), data wrangling (Narayan et al., 2022), and form processing (Chen et al., 2021). One key characteristic of these tasks is that they often require running LLM inference in batches over a large number of tokens (e.g., all the documents in a company’s

<sup>1</sup>Stanford University <sup>2</sup>UC Berkeley <sup>3</sup>ETH Zurich <sup>4</sup>Yandex  
<sup>5</sup>HSE University <sup>6</sup>Meta <sup>7</sup>Carnegie Mellon University. Correspondence to: Ying Sheng <ying1123@stanford.edu>.

Proceedings of the 40<sup>th</sup> International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

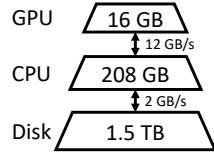
This version has an extended author list compared to the one archived in ICML.

corpus), and are less sensitive to latency. As a result, it is possible to trade off latency for higher throughput in these workloads, providing opportunities to reduce resource requirements.

Prior efforts to lower resource requirements of LLM inference correspond to three directions: (1) *model compression* to decrease total memory footprint (Dettmers et al., 2022; Yao et al., 2022; Frantar et al., 2022; Xiao et al., 2022); (2) *collaborative inference* to amortize inference cost via decentralization (Borzunov et al., 2022); and (3) *offloading* to utilize memory from CPU and disk (Aminabadi et al., 2022; HuggingFace, 2022). These techniques have significantly lowered the resource requirements for using LLMs, but there are distinct limitations. Research in the first two directions often assume that the model fits into the GPU memory and thereby struggle to run 175B-scale models with a single commodity GPU. On the other hand, state-of-the-art offloading-based systems in the third category do not achieve acceptable throughput on a single GPU due to inefficient I/O scheduling and tensor placement. For example, these systems can be bottlenecked by small batch sizes (e.g., batch sizes of only one or two for OPT-175B in some cases).

Our focus is designing efficient *offloading* strategies for high-throughput generative inference, *on a single commodity GPU*. To run an LLM with limited GPU memory, we can offload it to secondary storage and perform computation part-by-part by partially loading it. On a typical machine, there are three levels of the memory hierarchy, as illustrated in the figure to the right. Higher levels are faster but scarce, while lower levels are slower but abundant. In throughput-oriented scenarios, we can sacrifice latency by using a large batch size, and amortize the expensive I/O operations among different memory hierarchies over a large batch of inputs, overlapped with computation. Fig. 1 shows the latency-throughput trade-off of three inference systems with offloading on a single NVIDIA T4 (16 GB) GPU. Note that the performance in terms of latency and throughput on limited resources is significantly inferior to that of the cases with sufficient resources.

Achieving high-throughput generative inference with limited GPU memory is challenging even if we can sacrifice the latency. The first challenge is to design an *efficient offloading strategy*. During generative inference, there are three kinds of tensors: weights, activations, and key-value (KV) cache. The strategy should specify what tensors to offload, where to offload them within the three-level memory hierarchy, and when to offload them during inference. The batch-by-batch, token-by-token, and layer-by-layer structure of the computation forms a complex dependency graph



where there are multiple ways to conduct computation. Together, these choices form a complex design space. Existing offloading-based inference systems (Aminabadi et al., 2022; HuggingFace, 2022) inherit strategies from training, which turn out to be some suboptimal points for inference, performing excessive I/O and achieving throughput far below theoretical hardware limits.

The second challenge is to develop *effective compression strategies*. Previous works have demonstrated promising results in compressing the weights and activations of LLMs. However, when combining compression with offloading for high-throughput inference, the I/O costs and memory reduction of the weights and KV cache become more important, motivating alternative compression schemes.

To address these challenges, we present FlexGen, an offloading framework for high-throughput LLM inference. FlexGen aggregates memory from the GPU, CPU, and disk, and efficiently schedules I/O operations, along with possible compression methods and distributed pipeline parallelism.

**(Contribution 1)** We formally define a search space of possible offloading strategies by considering computation schedule, tensor placement, and computation delegation. We prove that our search space captures a computation order with I/O complexity within  $2\times$  of optimality. We then develop a linear programming-based search algorithm to optimize the throughput within the search space. This algorithm can be configured for various hardware specifications and can be easily extended to incorporate latency and throughput constraints, thus helping to navigate the trade-off space smoothly. Compared with existing strategies, our solution unifies the placement of weights, activations, and the KV cache, enabling a dramatically higher batch size upper bound, which is key to achieving high throughput.

**(Contribution 2)** We show that it is possible to compress both the weights and KV cache for LLMs like OPT-175B to 4 bits without retraining or calibration, all with negligible accuracy loss. This is achieved through fine-grained group-wise quantization (Shen et al., 2020), which is suitable for reducing I/O costs and memory usage during offloading.

**(Contribution 3)** We demonstrate the efficiency of FlexGen by running OPT-175B on NVIDIA T4 (16GB) GPUs. Compared to DeepSpeed Zero-Inference (Aminabadi et al., 2022) and Hugging Face Accelerate (HuggingFace, 2022), two state-of-the-art offloading-based inference systems, FlexGen often allows a batch size that is orders of magnitude larger. As a result, FlexGen can achieve much higher throughputs. On a single T4 GPU with 208 GB CPU DRAM and 1.5 TB SSD, input sequence length 512, and output sequence length 32:

- With the same latency of 5000 seconds, FlexGen (effective batch size 64, or 2048 tokens in total) can achieve

more than  $40\times$  higher throughput than DeepSpeed Zero-Inference (batch size 1, or 32 tokens in total), while Hugging Face Accelerate cannot complete a single batch.

- By allowing a higher latency of 12000 seconds, FlexGen achieves  $69\times$  higher maximum throughput compared to baselines because it can enlarge the effective batch size to 256 (8192 tokens generated in total), while DeepSpeed Zero-Inference and Hugging Face Accelerate cannot use a batch size larger than 2 due to out-of-memory issues.
- If allowing 4-bit compression, FlexGen can reach  $100\times$  higher maximum throughput with effective batch size 144 (4608 tokens generated in total) with latency 4000 seconds by holding all weights in CPU and getting rid of disk offloading.

We also compare offloading and decentralized collective inference based on FlexGen and Petals (Borzunov et al., 2022) as two representative systems. We conduct comparisons between the two systems from the aspects of delay and bandwidth of the decentralized network and output sequence length. The results show that FlexGen outperforms a decentralized Petals cluster in terms of per-GPU throughput and can even achieve lower latency in certain cases.

## 2. Related Work

Given the recent advances of LLMs, LLM inference has become an important workload, encouraging active research from both the **system** side and the **algorithm** side.

Recent years have witnessed the emergence of systems specialized for LLM inference, such as FasterTransformer (NVIDIA, 2022), Orca (Yu et al., 2022), LightSeq (Wang et al., 2021), PaLM inference (Pope et al., 2022), TurboTransformers (Fang et al., 2021), DeepSpeed Inference (Aminabadi et al., 2022), and Hugging Face Accelerate (HuggingFace, 2022). Unfortunately, most of these systems focus on latency-oriented scenarios with high-end accelerators, limiting their deployment for throughput-oriented inference on easily accessible hardware. To enable LLM inference on such commodity hardware, offloading is an essential technique — as far as we know, among current systems, only DeepSpeed Zero-Inference and Hugging Face Accelerate support offloading. These inference systems typically inherit the offloading techniques from training systems (Rajbhandari et al., 2021; Ren et al., 2021; Li et al., 2022; Huang et al., 2020; Wang et al., 2018) but ignore the special computational property of generative inference. They fail to exploit the structure of the throughput-oriented LLM inference computation and miss great opportunities for efficient scheduling of I/O traffic. Another attempt to enable LLM inference on accessible hardware is collaborative computing proposed by Petals (Borzunov et al., 2022).

There are also many algorithm-oriented works that relax certain aspects of computation in LLM inference to accelerate the computation or reduce the memory footprint. Both sparsification (Hoefler et al., 2021; Frantar & Alistarh, 2023) and quantization (Kwon et al., 2022; Yao et al., 2022; Park et al., 2022; Xiao et al., 2022; Frantar et al., 2022; Dettmers et al., 2022) have been adopted for LLM inference. On the quantization side, prior works have shown weights can be compressed down to 3 bits without compressing activations (Frantar et al., 2022), or both weights and activations can be compressed to 8 bits (Yao et al., 2022; Dettmers et al., 2022; Xiao et al., 2022). In FlexGen, we compress both the weights and KV cache to 4 bits and show how to combine the compression with offloading to make further improvements.

Within broader domains, memory optimizations and offloading have been studied for training (Huang et al., 2020; Ren et al., 2021; Steiner et al., 2022) and linear algebra (Jia-Wei & Kung, 1981; Demmel, 2013).

## 3. Background: LLM Inference

In this section, we describe the LLM inference workflow and its memory footprint.

**Generative Inference.** A typical LLM generative inference task consists of two stages: i) the *prefill* stage which takes a prompt sequence to generate the key-value cache (KV cache) for each transformer layer of the LLM; and ii) the *decoding* stage which utilizes and updates the KV cache to generate tokens step-by-step, where the current token generation depends on previously generated tokens.

For a particular inference computation, denote the batch size by  $b$ , the input sequence length by  $s$ , the output sequence length by  $n$ , the hidden dimension of the transformer by  $h_1$ , the hidden dimension of the second MLP layer by  $h_2$ , and the total number of transformer layers by  $l$ . Given the weight matrices of a transformer layer specified by  $\mathbf{w}_K^i, \mathbf{w}_Q^i, \mathbf{w}_V^i, \mathbf{w}_O^i, \mathbf{w}_1^i, \mathbf{w}_2^i$ , where  $\mathbf{w}_K^i, \mathbf{w}_Q^i, \mathbf{w}_V^i, \mathbf{w}_O^i \in \mathcal{R}^{h_1 \times h_1}, \mathbf{w}_1^i \in \mathcal{R}^{h_1 \times h_2}$ , and  $\mathbf{w}_2^i \in \mathcal{R}^{h_2 \times h_1}$ .

During the *prefill phase*, the input of the  $i$ -th layer is specified by  $\mathbf{x}^i$ , and key, value, query, and output of the attention layer is specified by  $\mathbf{x}_K^i, \mathbf{x}_V^i, \mathbf{x}_Q^i, \mathbf{x}_{\text{Out}}^i$ , where  $\mathbf{x}^i, \mathbf{x}_K^i, \mathbf{x}_V^i, \mathbf{x}_Q^i, \mathbf{x}_{\text{Out}}^i \in \mathcal{R}^{b \times s \times h_1}$ . Then, the cached key, value can be computed by:

$$\mathbf{x}_K^i = \mathbf{x}^i \cdot \mathbf{w}_K^i; \quad \mathbf{x}_V^i = \mathbf{x}^i \cdot \mathbf{w}_V^i$$

The rest of the computation in the  $i$ -th layer is:

$$\begin{aligned} \mathbf{x}_Q^i &= \mathbf{x}^i \cdot \mathbf{w}_Q^i \\ \mathbf{x}_{\text{Out}}^i &= f_{\text{Softmax}} \left( \frac{\mathbf{x}_Q^i \mathbf{x}_K^{i^T}}{\sqrt{h}} \right) \cdot \mathbf{x}_V^i \cdot \mathbf{w}_O^i + \mathbf{x}^i \\ \mathbf{x}^{i+1} &= f_{\text{relu}} (\mathbf{x}_{\text{Out}}^i \cdot \mathbf{w}_1) \cdot \mathbf{w}_2 + \mathbf{x}_{\text{Out}}^i \end{aligned}$$

During the *decode phase*, given  $\mathbf{t}^i \in \mathcal{R}^{b \times 1 \times h_1}$  as the embedding of the current generated token in the  $i$ -th layer, the inference computation needs to i) update the KV cache:

$$\begin{aligned}\mathbf{x}_K^i &\leftarrow \text{Concat}(\mathbf{x}_K^i, \mathbf{t}^i \cdot \mathbf{w}_K^i) \\ \mathbf{x}_V^i &\leftarrow \text{Concat}(\mathbf{x}_V^i, \mathbf{t}^i \cdot \mathbf{w}_V^i)\end{aligned}$$

and ii) compute the output of the current layer:

$$\begin{aligned}\mathbf{t}_Q^i &= \mathbf{t}^i \cdot \mathbf{w}_Q^i \\ \mathbf{t}_{\text{Out}}^i &= f_{\text{Softmax}}\left(\frac{\mathbf{t}_Q^i \mathbf{x}_K^i^T}{\sqrt{h}}\right) \cdot \mathbf{x}_V^i \cdot \mathbf{w}_O^i + \mathbf{t}^i \\ \mathbf{t}^{i+1} &= f_{\text{relu}}(\mathbf{t}_{\text{Out}}^i \cdot \mathbf{w}_1) \cdot \mathbf{w}_2 + \mathbf{t}_{\text{Out}}^i\end{aligned}$$

**Memory Analysis.** The memory footprint of LLM inference mainly comes from the model weights and the KV cache. Considering the OPT-175B model in FP16, the total number of bytes to store the parameters can be roughly <sup>1</sup> calculated by  $l(8h_1^2 + 4h_1h_2)$ . The total number of bytes to store the KV cache in peak is  $4 \times blh_1(s+n)$ .

In a realistic setting with a sufficient number of GPUs, the OPT-175B model ( $l = 96, h_1 = 12288, h_2 = 49152$ ) takes 325 GB. With a batch size of  $b = 512$ , an input sequence length  $s = 512$ , and an output sequence length of  $n = 32$ , the total memory required to store the KV cache is 1.2 TB, which is  $3.8 \times$  the model weights, making the KV cache a new bottleneck of large-batch high-throughput inference. In FlexGen, for OPT-175B, we enlarge the effective batch size to 256 to achieve the throughput at 0.69 token/s.

**Throughput and Latency.** Considering an effective batch size  $b$ , an input sequence length  $s$ , and an output sequence length of  $n$ , the latency  $t$  is defined as the total number of seconds spent to process the prompts and generate all the  $bn$  tokens. The generation throughput is defined as  $bn/t$ .

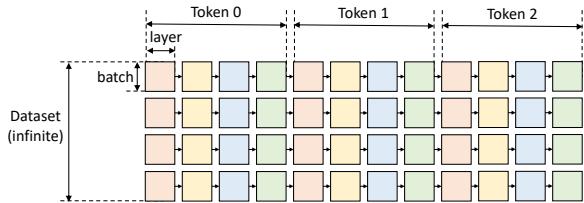


Figure 2. Computational graph of LLM inference.

## 4. Offloading Strategy

In this section, we do not relax any computation of LLM inference and illustrate how to formalize the offloading procedure under the GPU, CPU, and disk memory hierarchy. We first formulate the problem and then construct the search space of the possible offloading strategies in FlexGen. To find an efficient strategy, FlexGen builds an analytical cost model and searches for configurations with an optimizer based on linear programming.

<sup>1</sup>We ignore the embedding layer(s), which is relatively small.

### 4.1. Problem Formulation

Consider a machine with three devices: a GPU, a CPU, and a disk. The GPU and CPU can perform computation while the disk cannot. The three devices form a three-level memory hierarchy where the GPU has the smallest but fastest memory and the disk has the largest but slowest memory. When an LLM cannot fit entirely within the GPU, we need to offload it to secondary storage and perform computation part-by-part by partially loading the LLM.

We formulate the generative inference with offloading as a graph traversal problem. Fig. 2 shows an example computational graph, where the model has 4 layers and we generate 3 tokens per prompt. As our focus is throughput-oriented scenarios, we assume a given dataset with an infinite number of prompts that need to be processed. In the figure, a square means the computation of a GPU batch for a layer. The squares with the same color share the same layer weights. We define a valid path as a path that traverses (i.e., computes) all squares, while subject to the following constraints:

- A square can only be computed if all squares to its left on the same row were computed.
- To compute a square on a device, all its inputs (weights, activations, cache) must be loaded to the same device.
- After being computed, a square produces two outputs: activations and KV cache. The activations should be stored until its right sibling is computed. The KV cache should be stored until the rightmost square on the same row is computed.
- At any time, the total size of tensors stored on a device cannot exceed its memory capacity.

The goal is to find a valid path that minimizes the total execution time, which includes the compute cost and I/O cost when moving tensors between devices.

### 4.2. Search Space

Given the formulation above, we construct a search space for possible valid strategies in FlexGen.

**Compute schedule.** Intuitively, there are two orders to traverse the graph in Fig. 2: row-by-row and column-by-column. All existing systems (Aminabadi et al., 2022; HuggingFace, 2022) traverse the graph row-by-row, as shown in Fig. 3(a). This is reasonable because it is the fastest way to finish the generation for one batch and the KV cache can be freed immediately after a row. However, because every two contiguous squares do not share weights, this schedule has to repeatedly load the weights and incurs huge I/O costs.

To reduce the I/O costs of the weights, we can traverse the graph column-by-column. All squares in a column share weights, so we can let the weights stay on GPU for reusing and only load/unload the activations and KV cache. How-

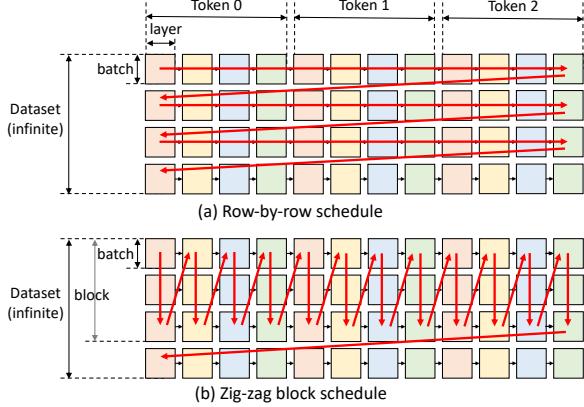


Figure 3. Two different schedules. The red arrows denote the computation order.

#### Algorithm 1 Block Schedule with Overlapping

```

for  $i = 1$  to  $\text{generation\_length}$  do
    for  $j = 1$  to  $\text{num\_layers}$  do
        // Compute a block with multiple GPU batches
        for  $k = 1$  to  $\text{num\_GPU\_batches}$  do
            // Load the weight of the next layer
             $\text{load\_weight}(i, j + 1, k)$ 
            // Store the cache and activation of the prev batch
             $\text{store\_activation}(i, j, k - 1)$ 
             $\text{store\_cache}(i, j, k - 1)$ 
            // Load the cache and activation of the next batch
             $\text{load\_cache}(i, j, k + 1)$ 
             $\text{load\_activation}(i, j, k + 1)$ 
            // Compute this batch
             $\text{compute}(i, j, k)$ 
            // Synchronize all devices
             $\text{synchronize}()$ 
        end for
    end for
end for

```

ever, we cannot traverse a column all the way to the end because the activations and KV cache still need to be stored. Hence, we have to stop when they fill the CPU and disk memory. Taking all this into consideration, we converge to a zig-zag block schedule, as shown in Fig. 3(b). Besides, we propose another more advanced and I/O-optimal schedule, but only implement the simpler block schedule due to the practical implementation difficulty of the optimal one. However, we prove that the block schedule is at most twice worse than the optimal schedule in Appendix A.2.

**Theorem 4.1.** *The I/O complexity of the zig-zag block schedule is within  $2 \times$  of the optimal solution.*

Another typical optimization is overlapping. We can overlap the weights load of the next layer, cache/activation load of the next batch, cache/activation store of the previous batch, and the computation of the current batch. Adding overlapping to the block schedule results in Algorithm 1. The first six functions in the innermost loop can be seen as launched

in parallel with six logical threads because there are no dependencies. The last function then synchronizes these six logical threads. We rely on operating systems and CUDA drivers to resolve the schedule of the underlying hardware resources. As a conclusion, the algorithm introduces two parameters into our search space: the GPU batch size and the number of GPU batches in a block. The product of the GPU batch size and the number of GPU batches is called block size (or **effective batch size**).

**Tensor placement.** Besides compute schedule, a strategy should specify how to store these tensors within the memory hierarchy. We use three variables  $wg$ ,  $wc$ , and  $wd$  to define the percentages of weights stored on GPU, CPU, and disk respectively. Similarly, we use three variables  $hg$ ,  $hc$ ,  $hd$  to define the percentages of activations and use  $cg$ ,  $cc$ ,  $cd$  for the KV cache. Given the percentages, there are still multiple ways to partition the tensors. Taking weight tensors as an example, from coarse grain to fine grain, we can partition the weights at the model granularity (e.g., assign 50% of the layers in a model to the GPU), at the layer granularity (e.g., assign 50% of the tensors in a layer to the GPU), or at the tensor granularity (e.g., assign 50% of the elements in a tensor to the GPU). Coarser granularity leads to lower runtime overhead but it is less flexible and its cost is difficult to analyze. Considering both the runtime overhead and desired flexibility, we use layer granularity for weights, and tensor granularity for activations and the KV cache.

**Computation delegation.** While CPUs are much slower than GPUs, we find using CPU compute can still be beneficial in some cases. This is because the computation of attention scores during decoding is I/O-bounded. Consider a case where the KV cache is stored on the CPU. Computing the attention scores on the GPU requires moving the entire KV cache to the GPU, which incurs a substantial I/O cost as the KV cache is huge. In contrast, computing the attention score on the CPU does not require moving the KV cache. It only requires moving the activations from the GPU to the CPU. Quantitatively, let  $b$  be the GPU batch size,  $s$  be the sequence length, and  $h_1$  be the hidden size. The size of the moved KV cache is  $b \times s \times h_1 \times 4$  bytes, and the size of the moved activation is  $b \times h_1 \times 4$  bytes, so computing attention score on CPU reduces I/O by  $s \times$ . For long sequences (e.g.,  $s \geq 512$ ), it is better to compute the attention scores on the CPU if the associated KV cache is not stored on the GPU.

#### 4.3. Cost Model and Policy Search

The schedule and placement in Section 4.2 constructs a search space with several parameters. Now we develop an analytical cost model to estimate the execution time given these algorithm parameters and hardware specifications.

**Cost Model.** The cost model predicts the latency during prefill for one layer denoted as  $T_{\text{pre}}$ , and the averaged la-

tency during decoding for one layer denoted as  $T_{gen}$  in one block. The total latency for computing a block can then be estimated as  $T = T_{pre} \cdot l + T_{gen} \cdot (n - 1) \cdot l$ , where  $l$  is the number of layers and  $n$  is the number of tokens to generate.

Assuming perfect overlapping,  $T_{pre}$  can be estimated as  $T_{pre} = \max(ctog^p, gtoc^p, dtoc^p, ctod^p, comp^p)$ , where  $ctog^p, gtoc^p, dtoc^p, ctod^p, comp^p$  denote the latency of read from CPU to GPU, write from GPU to CPU, read from disk to CPU, write from CPU to disk, computation, respectively, during prefill for one layer.

Similarly,  $T_{gen}$  can be estimated as  $T_{gen} = \max(ctog^g, gtoc^g, dtoc^g, ctod^g, comp^g)$ , with  $ctog^g, gtoc^g, dtoc^g, ctod^g, comp^g$  denoting the latency of read from CPU to GPU, write from GPU to CPU, read from disk to CPU, write from CPU to disk, computation, respectively, during decoding for one layer.

For I/O terms like  $dtoc^g$ , it is estimated by summing up the I/O events, which contain weights, activations, and cache reads. The size of FP16 weights for one transformer layer is  $8h_1^2 + 4h_1 \cdot h_2$  bytes, with  $h_1$  denoting the hidden size, and  $h_2$  denoting the hidden size of the second MLP layer. Let  $bls$  be the block size and  $s$  be the prompt length; then the size of activations for one layer is  $2 \cdot bls \cdot h_1$ . The size of the KV cache for one layer on average is  $4 \cdot bls \cdot (s + \frac{n}{2}) \cdot h_1$ . We have to load  $wd, hd, cd$  percent of weights, activations, and the KV cache from the disk respectively so that the total latency of disk read is  $dtoc^g = \frac{1}{\text{disk.to.cpu.bandwidth}} ((8h_1^2 + 4h_1 \cdot h_2) \cdot wd + 4 \cdot bls \cdot (s + \frac{n}{2}) \cdot h_1 \cdot cd + 2 \cdot bls \cdot h_1 \cdot hd)$ .

Similarly for computation terms, we sum up all computation events, including matrix multiplications and batched matrix multiplications on the CPU and the GPU.

Besides latency estimation, we also estimate the peak memory usage of the GPU, CPU, and disk, and then we add memory constraints. The full cost model is in Appendix A.3.

**Policy Search.** A policy includes 11 variables: block size  $bls$ , GPU batch size  $gbs$ , weight placement  $wg, wc, wd$ , activation placement  $hg, hc, hd$ , and KV cache placement  $cg, cc, cd$ . In practice, the percentage cannot be an arbitrary real number between 0 and 1, because the tensor cannot be split arbitrarily. However, we relax the percentage variables in the cost model to be any real number between 0 and 1 since it is changing gradually. We solve the problem as a two-level optimization problem. We first enumerate a few choices of  $(bls, gbs)$  tuple. Typically,  $gbs$  is a multiple of 4, and  $bls$  is less than 20 so there are not too many choices. Then with the fixed  $bls, gbs$ , finding the best placement  $p = (wg, wc, wd, cg, cc, cd, hg, hc, hd)$  becomes a linear programming problem shown in Eq. (1). The linear programming problem can be solved very quickly because there are only 9 variables. This formulation can also be flexibly extended to include latency constraints and model

approximate methods such as compression.

$$\begin{aligned} \min_p \quad & T / bls \\ \text{s.t.} \quad & gpu \text{ peak memory} < gpu \text{ mem capacity} \\ & cpu \text{ peak memory} < cpu \text{ mem capacity} \\ & disk \text{ peak memory} < disk \text{ mem capacity} \\ & wg + wc + wd = 1 \\ & cg + cc + cd = 1 \\ & hg + hc + hd = 1 \end{aligned} \quad (1)$$

To use the cost model, we run profiling on the hardware to sample some data points and fit the hardware parameters. We then call the optimizer to get an offloading policy. Due to our relaxation and the hardness of accurately modeling peak memory usage (e.g., fragmentation), sometimes a strategy from the policy search can run out of memory. In this case, we manually adjust the policy slightly. The cost model can usually return a good policy, but it is common that a better policy can be obtained by tuning manually.

#### 4.4. Extension to Multiple GPUs

We discuss how to extend the offloading strategy in FlexGen if there are multiple GPUs. Although we can find a nearly optimal strategy for one GPU, the strategy is still heavily limited by I/O and has a low GPU utilization. If we are given more GPUs and more CPUs, model parallelism can be utilized to reduce the memory pressure of each GPU, which can potentially lead to a super-linear scaling in decoding.

There are two kinds of model parallelisms: tensor and pipeline parallelism (Narayanan et al., 2021; Zheng et al., 2022). Tensor parallelism can reduce the single-query latency but pipeline parallelism can achieve good scaling on throughput due to its low communication costs. Since we target throughput, FlexGen implements pipeline parallelism.

We use pipeline parallelism by equally partitioning an  $l$ -layer LLM on  $m$  GPUs, and then the execution of all GPUs follows the same pattern. The problem is reduced to running an  $n/m$ -layer transformer on one GPU. We can directly reuse the policy search developed for one GPU. To achieve micro-batch pipelining, a new for-loop is added to Algorithm 1 to combine the iteration-level pipeline parallel execution schedule (Huang et al., 2019; Yu et al., 2022) with our single-device offloading runtime.

## 5. Approximate Methods

The previous section focuses on the exact computation. However, the inference throughput can be greatly boosted with negligible accuracy loss by allowing some approximations, because LLMs are typically robust to careful approximations. This section introduces two such approximations: group-wise quantization and sparse attention.

**Group-wise Quantization.** We show that both the weights and KV cache can be directly quantized into 4-bit integers without any retraining or calibration on OPT-175B, all while preserving similar accuracy (Section 6.2). When compared to some related works (Yao et al., 2022; Dettmers et al., 2022; Xiao et al., 2022) that try to use integer matrix multiplication mainly for accelerated computation, the goal of quantization in our case is primarily for compression and reducing I/O costs. Therefore, we can choose a fine-grained quantization format in favor of a high compression ratio and dequantize the tensors back to FP16 before computation. We use a fine-grained group-wise asymmetric quantization method (Shen et al., 2020). Given a tensor, we choose  $g$  contiguous elements along a certain dimension as a group. For each group, we compute the *min* and *max* of the group elements and quantize each element  $x$  into  $b$ -bit integers by  $x_{\text{quant}} = \text{round} \left( \frac{x - \text{min}}{\text{max} - \text{min}} \times (2^b - 1) \right)$ .

The tensors are stored in the quantized format and converted back to FP16 before computation. Since both the weights and KV cache consume a significant amount of memory, we compress both to 4 bits with a group size of 64. There are multiple ways to choose which dimension to group on. We find that grouping the weights along the output channel dimension and the KV cache along the hidden dimension preserves the accuracy while being runtime-efficient in practice. One thing to mention is that such a fine-grained group-wise quantization in FlexGen causes some overhead in compression and decompression. Such an overhead could be very significant if run on a CPU which makes the CPU delegation useless, so we turn off the CPU delegation when enabling quantization. A concurrent work (Dettmers & Zettlemoyer, 2022) also finds that 4-bit precision is almost optimal for total model bits and zero-shot accuracy on OPT models. Compared to this previous work, we first propose to compress the KV cache and present the results on OPT-175B.

**Sparse Attention.** We demonstrate that the sparsity of self-attention can be exploited by only loading the top 10% attention value cache on OPT-175B, all while maintaining the model quality. We present one simple Top-K sparse approximation. After computing the attention matrices, for each query, we calculate the indices of its Top-K tokens from the K cache. We then simply drop the other tokens and only load a subset of the V cache according to the indices.

The application of these approximations is straightforward. We present these preliminary but interesting results and intend to emphasize that FlexGen is a general framework that can seamlessly plug in many approximation methods.

## 6. Evaluation

**Hardware.** We run experiments on the NVIDIA T4 GPU instances from Google Cloud. The hardware specifications are

Table 1. Hardware Specs

Device	Model	Memory
GPU	NVIDIA T4	16 GB
CPU	Intel Xeon @ 2.00GHz	208 GB
Disk	Cloud default SSD (NVMe)	1.5 TB

listed in Table 1. The read bandwidth of SSD is about 2GB/s and the write bandwidth is about 1GB/s. Our methods and implementations do not depend on specific hardware architectures. Some architecture (e.g. unified memory) could be more friendly to our method. See Appendix A.4 for discussions and experiments on different hardware setups.

**Model.** OPT models (Zhang et al., 2022) with 6.7B to 175B parameters are used in the evaluation. Although we do not evaluate other models, the offloading in FlexGen can be applied to other transformer LLMs, e.g., GPT-3 (Brown et al., 2020), PaLM (Chowdhery et al., 2022), and BLOOM (Scao et al., 2022) because they all share a similar structure.

**Workload.** Our focus is high-throughput generation on a given dataset. We use synthetic datasets where all prompts are padded to the same length. The system is required to generate 32 tokens for each prompt. We test two prompt lengths: 512 and 1024 (for experiments in more settings, see Appendix A.4). The evaluation metric is generation throughput, defined as the number of generated tokens / (prefill time + decoding time). Sometimes running a full batch takes too long for certain systems — in this cases, we generate fewer tokens and project the final throughput. We use dummy model weights in throughput benchmarks for all systems and real weights for accuracy evaluations.

**Baseline.** We use DeepSpeed ZeRO-Inference (Aminabadi et al., 2022) and Hugging Face Accelerate (HuggingFace, 2022) as baselines. They are the only systems that can run LLMs with offloading when there is not enough GPU memory. DeepSpeed supports offloading the whole weights to the CPU or disk. It uses ZeRO data parallelism if there are multiple GPUs. Accelerate supports offloading a fraction of the weights. It does not support distributed GPUs on different machines. Both of them use the row-by-row schedule and can only put cache/activations on GPU. These systems support different quantization methods. However, the quantization in Accelerate is not compatible with offloading, and the quantization in DeepSpeed cannot preserve accuracy up to 175B, so we do not enable quantization on these systems. In addition to offloading, decentralized collaborative inference is another option to lower the resource requirement for LLM inference. Thus, we also include Petals (Borzunov et al., 2022; Ryabinin et al., 2023) as an additional baseline.

**Implementation.** FlexGen is implemented on top of PyTorch (Paszke et al., 2019). FlexGen manages multiple CUDA streams and CPU threads to overlap I/O with compute. FlexGen creates files for tensors stored on the disk and maps them as virtual memory to access them.

## 6.1. Offloading

**Maximum throughput benchmark.** We first evaluate the maximum generation throughput the systems can achieve with one GPU on two prompt lengths. As shown in Table 2, FlexGen outperforms all baselines in all cases. On OPT-6.7B, Accelerate and FlexGen can successfully fit the whole model into a single GPU, so they choose to only use the GPU. DeepSpeed has a higher memory overhead and cannot fit OPT-6.7B into the GPU, so it uses slower CPU offloading. On OPT-30B, all systems switch to CPU offloading. DeepSpeed and Accelerate store the KV cache on the GPU, so they cannot use a very large batch size, while FlexGen offloads most weights and all KV cache to the CPU and enables a larger GPU batch size. In addition, FlexGen reuses the weights by block scheduling. On OPT-175B, all systems start to offload the weights to the disk. Baseline systems can only use a maximum batch size of 2, but FlexGen can use a GPU batch size of 32 and a block size of  $32 \times 8$ , achieving a  $69\times$  higher throughput. With compression enabled, FlexGen achieves a  $112\times$  higher generation throughput on a single GPU for prompt sequence length 512. This huge improvement is because FlexGen uses an effective batch size of 144 and compresses the weights and KV cache to fit into CPU memory to avoid slow disk swapping. More details on the policy setups and effective batch sizes can be found in Appendix A.4. More experiments on how disk specification affects the throughput see Appendix A.4.

Table 3 shows the results on 4 machines, with one GPU on each machine. OPT-30B or OPT-175B still cannot fit into 4 GPUs. Naively, we can run 4 independent FlexGen in a data-parallel fashion to get a linear scaling on throughput. But here we show that pipeline parallelism can achieve super-linear scaling on decoding throughput. With pipeline parallelism, the memory pressure of each machine is reduced so we can switch from small batch sizes to larger batch sizes, or switch from disk offloading to CPU-only offloading. In Table 3, FlexGen does not achieve linear scaling on generation throughput (which counts both prefill and decoding time costs). This is because there are pipeline bubbles during the prefill stage and our workload settings only generate 32 tokens. However, FlexGen achieves super-linear scaling on decoding throughput (which only counts decoding time costs assuming the prefill is done). This means if we generate more tokens, pipeline parallelism will show its benefits as decoding time will dominate.

**Latency-throughput trade-off.** We configure these systems to achieve maximum throughput under various latency constraints and draw their latency-throughput trade-off curves in Fig. 1. FlexGen sets a new Pareto-optimal frontier that significantly outperforms baselines. On the low-latency side, FlexGen supports partial offloading and uses more space for weights. On the high-throughput side,

Table 2. Generation throughput (token/s) of different systems. Accelerate, DeepSpeed, and FlexGen use 1 GPU. Petals uses 1 GPU for OPT-6.7B, 4 GPUs for OPT-30B, and 24 GPUs for OPT-175B, but reports per-GPU throughput. We benchmark Petals under a good network assumption with a delay of less than 10ms and bandwidth of 1 Gbps. The models are run in INT8 as the default for Petals. See Section 6.3 for more details about Petals. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. “OOM” means out-of-memory.

Seq. length	512			1024		
	6.7B	30B	175B	6.7B	30B	175B
Accelerate	25.12	0.62	0.01	13.01	0.31	0.01
DeepSpeed	9.28	0.60	0.01	4.59	0.29	OOM
Petals	8.25	2.84	0.08	6.56	1.51	0.06
FlexGen	25.26	7.32	0.69	13.72	3.50	0.35
FlexGen (c)	29.12	8.70	1.12	13.18	3.98	0.42

Table 3. The scaling performance on 4 GPUs. The prompt sequence length is 512. The number of GPUs is denoted in the parenthesis. Generation throughput (token/s) counts the time cost of both prefill and decoding while decoding throughput only counts the time cost of decoding assuming prefill is done.

Metric	Generation Throughput			Decoding Throughput		
	6.7B	30B	175B	6.7B	30B	175B
FlexGen (1)	25.26	7.32	0.69	38.28	11.52	0.83
FlexGen (4)	201.12	23.61	2.33	764.65	48.94	3.86
DeepSpeed (4)	50.00	6.40	0.05	50.20	6.40	0.05

FlexGen aggressively offloads all things out of the GPU to achieve a large GPU batch size and block size. Given the same latency requirement of 5000 seconds, FlexGen without compression can achieve a  $40\times$  higher throughput compared to DeepSpeed and Accelerate. If allowing a higher latency and compression, FlexGen can further boost throughput and reach a  $100\times$  improvement by using an effective batch size of 144. In this case, compression enables FlexGen to fit all things in the CPU memory and avoid disk I/O. The detailed latency, throughput, and policy setup can be found in Appendix A.4.

**Runtime breakdown.** We show the runtime breakdown of OPT-175B on FlexGen in Table 8 in Appendix A.4. We disable overlapping and profile the time used for major components. The GPU compute utilization is 82% and 13% for prefill and decoding, respectively.

**Ablation study.** We then isolate the improvement brought by each individual technique. Table 4 lists the throughput FlexGen can achieve if disabling one technique at a time. On OPT-30B, with all optimizations enabled, we put 20% weights on GPU, 80% weights on CPU, and all activations and KV cache to CPU. We also choose a GPU batch size of 48 and a block size of  $48 \times 3$ . “No policy search” illustrates the performance of worse strategies, showing the importance of a good policy. On both models, using CPU compute and overlapping brings non-trivial improvement. We also

**Table 4.** Ablation study of proposed techniques. The numbers are generation throughput on 1 GPU with prompt length 512. The gray tuple denotes a policy (GPU batch size  $\times$  #GPU-batch,  $w_g$ ,  $w_c$ ). More see Appendix A.4.

Model size	30B	175B
All optimizations	7.32 (48 $\times$ 3, 20, 80)	0.69 (32 $\times$ 8, 0, 50)
No policy search	7.26 (48 $\times$ 3, 0, 100)	0.27 (32 $\times$ 1, 0, 50)
No overlapping	5.86	0.59
No CPU compute	4.03	0.62
No disk	7.32	OOM
w/ DeepSpeed policy	1.57	0.01

**Table 5.** The accuracy (higher is better) and perplexity (lower is better) with approximate methods.

Dataset	Lambada (acc)			WikiText (pppl)		
	Config	FP16	4-bit	4-bit-S	FP16	4-bit
OPT-30B	0.725	0.724	0.718	12.72	12.90	12.90
OPT-175B	0.758	0.756	0.756	10.82	10.94	10.94

port the policy used in DeepSpeed/Accelerate into FlexGen runtime, showing the suboptimality of their policy. A more detailed ablation study can be found in Appendix A.4.

**HELM and Data wrangling.** We tested the interaction of FlexGen and HELM (Liang et al., 2022) by evaluating a new model OPT-IML-30B (Iyer et al., 2022), which has not been included in the official release of HELM. FlexGen finishes the benchmark of 7 representative sub-scenarios in 21 hours, with all system overhead included, under the hardware setup described in Table 1. Table 9 in Appendix A.4 shows the details of the tasks and the corresponding running time. We also use FlexGen to run the data wrangling tasks (Narayan et al., 2022) with OPT models. The detailed task configurations and running time are in Appendix A.4.

## 6.2. Approximations

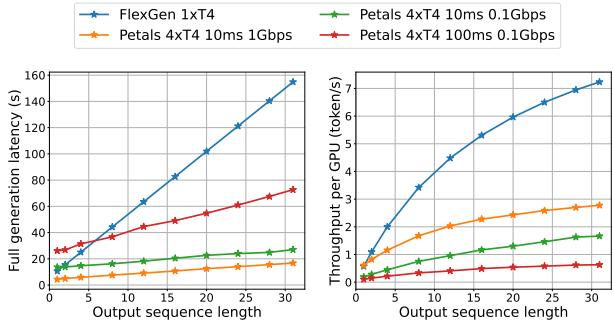
We use two tasks to show that our approximation methods exhibit negligible accuracy loss: next-word prediction on Lambada (Paperno et al., 2016) and language modeling on WikiText (Merity et al., 2016). As shown in Table 5, “4-bit” means using group-wise quantization to compress both weights and KV cache into 4-bit integers. “4-bit-S” means combining the quantization and sparse attention with a 10% sparsity on the value cache. Both methods show negligible accuracy loss compared to FP16. The results reveal the robustness of LLMs against these approximations. We also tried 3-bit compression but it cannot preserve accuracy.

## 6.3. Offloading vs. Collaborative Inference

We compare FlexGen and Petals under different network conditions by setting a private Petals cluster on GCP with 4 nodes having one T4 GPU per node. We use Linux traffic control to constrain the connections between instances to simulate a realistic decentralized network and benchmark the performance of an OPT-30B model (input sequence length: 512, output sequence length: 32). We tune the batch

size of each request to be 2 and issue requests by 6 parallel client processes to achieve the maximum throughput<sup>2</sup>. In addition, we normalize the throughput of Petals by the number of used GPUs. As shown in Fig. 4, we find that the throughput of FlexGen with a single T4 outperforms the per-GPU throughput of the Petals cluster under all tested network conditions. Petals does not utilize offloading, so it cannot use a very large batch size, which limits its scaling on throughput. Thus, we believe offloading could be a more efficient solution for throughput than communicating a large volume of activations in a long decentralized pipeline; on the other hand, collaborative inference can be a more viable option in more latency-sensitive scenarios.

Interestingly, we find that FlexGen can achieve lower latency than Petals in slow networks with short generation. We speculate this is because the network bandwidth becomes the bottleneck for activation transfer, and a large delay incurs a significant overhead on each communication step in the pipeline. For the curve of a 100ms delay network, we can observe a cross point between FlexGen and Petals. This is because the activations during prefill are larger than the activations during decoding by a factor of the input sequence length. Thus, the communication overhead is proportionally larger, which significantly slows down Petals during prefill.



**Figure 4.** Full latency and per-GPU throughput of FlexGen and Petals in different network delay and bandwidth.

## 7. Conclusion

We introduce FlexGen, a high-throughput generation engine for LLM inference, which focuses on latency-insensitive batch-processing tasks for resource-constrained scenarios.

## Acknowledgements

We would like to thank Clark Barrett and Joseph E. Gonzalez for funding support, and Zhiqiang Xie, Daniel Y. Fu, Hao Zhang, Nick Chow, Benjamin Spector, Guangxuan Xiao, Jue Wang, Arjun Desai, Yao Fu, Anjiang Wei, and Zihao Ye for their insightful review and discussions.

<sup>2</sup>The batch size of 1 did not result in a noticeably better latency.

## References

- Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., et al. Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 646–660. IEEE Computer Society, 2022.
- Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselet, A., Brunskill, E., et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.
- Borzunov, A., Baranchuk, D., Dettmers, T., Ryabinin, M., Belkada, Y., Chumachenko, A., Samygin, P., and Raffel, C. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188*, 2022.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.
- Chen, X., Maniatis, P., Singh, R., Sutton, C., Dai, H., Lin, M., and Zhou, D. Spreadsheetcoder: Formula prediction from semi-structured context. In *International Conference on Machine Learning*, pp. 1661–1672. PMLR, 2021.
- Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.
- Demmel, J. Communication-avoiding algorithms for linear algebra and beyond. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 585–585. IEEE, 2013.
- Dettmers, T. and Zettlemoyer, L. The case for 4-bit precision: k-bit inference scaling laws. *arXiv preprint arXiv:2212.09720*, 2022.
- Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. Llm.int8(): 8-bit matrix multiplication for transformers at scale. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022.
- Fang, J., Yu, Y., Zhao, C., and Zhou, J. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 389–402, 2021.
- Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.
- Frantar, E. and Alistarh, D. Massive language models can be accurately pruned in one-shot. *arXiv preprint arXiv:2301.00774*, 2023.
- Frantar, E., Ashkboos, S., Hoefer, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.
- Hoefer, T., Alistarh, D., Ben-Nun, T., Dryden, N., and Peste, A. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.*, 22(241):1–124, 2021.
- Huang, C.-C., Jin, G., and Li, J. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1341–1355, 2020.
- Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.
- HuggingFace. Hugging face accelerate. <https://huggingface.co/docs/accelerate/index>, 2022.
- Iyer, S., Lin, X. V., Pasunuru, R., Mihaylov, T., Simig, D., Yu, P., Shuster, K., Wang, T., Liu, Q., Koura, P. S., et al. Opt-iml: Scaling language model instruction meta learning through the lens of generalization. *arXiv preprint arXiv:2212.12017*, 2022.
- Jia-Wei, H. and Kung, H.-T. I/o complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pp. 326–333, 1981.
- Kwon, S. J., Kim, J., Bae, J., Yoo, K. M., Kim, J.-H., Park, B., Kim, B., Ha, J.-W., Sung, N., and Lee, D. Alphatuning: Quantization-aware parameter-efficient adaptation of large-scale pre-trained language models. *arXiv preprint arXiv:2210.03858*, 2022.
- Li, Y., Phanishayee, A., Murray, D., Tarnawski, J., and Kim, N. S. Harmony: Overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers. *arXiv preprint arXiv:2202.01306*, 2022.

- Liang, P., Bommasani, R., Lee, T., Tsipras, D., Soylu, D., Yasunaga, M., Zhang, Y., Narayanan, D., Wu, Y., Kumar, A., et al. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022.
- Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.
- Morton, A. Pagecachemangement. <https://code.google.com/archive/p/pagecache-mangagement/source/default/source>, 2008.
- Narayan, A., Chami, I., Orr, L., and Ré, C. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911*, 2022.
- Narayan, S., Cohen, S. B., and Lapata, M. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. *arXiv preprint arXiv:1808.08745*, 2018.
- Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.
- NVIDIA. Fastertransformer. <https://github.com/NVIDIA/FasterTransformer>, 2022.
- Paperno, D., Kruszewski, G., Lazaridou, A., Pham, N.-Q., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., and Fernández, R. The lambada dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1525–1534, 2016.
- Park, G., Park, B., Kwon, S. J., Kim, B., Lee, Y., and Lee, D. nuqmm: Quantized matmul for efficient inference of large-scale generative language models. *arXiv preprint arXiv:2206.09557*, 2022.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.
- Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*, 2022.
- Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.
- Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. Zero-offload: Democratizing billion-scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 551–564, 2021.
- Ryabinin, M., Dettmers, T., Diskin, M., and Borzunov, A. Swarm parallelism: Training large models can be surprisingly communication-efficient. *arXiv preprint arXiv:2301.11913*, 2023.
- Scao, T. L., Fan, A., Akiki, C., Pavlick, E., Ilić, S., Hesslow, D., Castagné, R., Luccioni, A. S., Yvon, F., Gallé, M., et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.
- Shen, S., Dong, Z., Ye, J., Ma, L., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 8815–8821, 2020.
- Steiner, B., Elhoushi, M., Kahn, J., and Hegarty, J. Olla: Optimizing the lifetime and location of arrays to reduce the memory usage of neural networks. 2022. doi: 10.48550/arXiv.2210.12924.
- Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S. L., Xu, Z., and Kraska, T. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pp. 41–53, 2018.
- Wang, X., Xiong, Y., Wei, Y., Wang, M., and Li, L. Lightseq: A high performance inference library for transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*, pp. 113–120, 2021.
- Xiao, G., Lin, J., Seznec, M., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*, 2022.
- Yao, Z., Aminabadi, R. Y., Zhang, M., Wu, X., Li, C., and He, Y. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Gonzalez, J. E., et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

## A. Appendix

### A.1. Notations

We use notations in Table 6 in this appendix.

Var	Meaning
$l$	number of layers in the model
$s$	prompt sequence length
$n$	output sequence length
$bls$	block size
$h_1$	hidden size
$h_2$	hidden size of the second MLP layer
$nh$	number of head in the model

Table 6. Notations

### A.2. Compute Schedule Optimality

This subsection discusses the graph traversal problem described in Section 4.1 and only considers the case that the model cannot fit in a single GPU. We assume no application of CPU computation. To compute a square, the GPU loads the tensors it needs and offloads the cache and activations when finished. We will analyze two schedules: the zig-zag block schedule used in Section 4.2 and an I/O-optimal diagonal block schedule introduced in this section. Note that our analysis only considers the theoretical I/O complexity. In the real system, the latency and memory consumption cannot be the same as in the theoretical calculations.

There are three things that need to be stored during the generation process: weights, activations, and the KV cache. From the computational graph, we have three observations. (1) Suppose we need to swap the weights in and out of the GPU. Whatever the portion is, to finish the generation for one prompt, we need to swap  $n$  times for  $n$  tokens. Therefore, it would be preferable to reuse the loaded weights for a batch of prompts, amortizing the weights I/O time. (2) Each square will output activations which will be fed into the next layer. Each row in the computational graph only needs to hold activations for one square at the same time. (3) For each square besides the last  $l$  squares in a row, the KV cache dumped by the square cannot be released until generating the last token (the last  $l$  columns in the computational graph). It is not shared across rows or columns, which will be the major factor in limiting the batch size.

#### A.2.1. ZIG-ZAG BLOCK SCHEDULE AND DIAGONAL BLOCK SCHEDULE

**Zig-zag block schedule.** Inspired by the three observations introduced in Section 4.2, we compute the first column in the computational graph for  $bls$  samples, save the dumped caches and activations, then compute the second column for  $bls$  samples, until the last column for  $bls$  samples. We

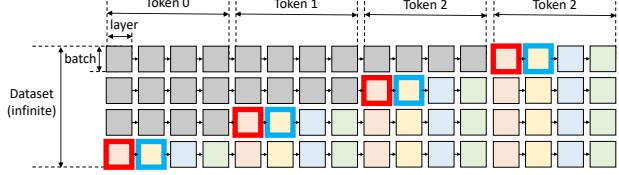


Figure 5. diagonal block schedule

call  $bls$  as the block size as introduced in Section 4.2. The computed  $bls \cdot n \cdot l$  squares are called a block.

Assume FP16 precision, to generate  $n \cdot bls$  tokens during one block computation, we have to load  $n$  times the whole model weights, do I/O operations on activations with  $2(2h_1 \cdot s \cdot bls \cdot l + 2h_1 \cdot bls \cdot l \cdot (n - 1))$  bytes in total, and do I/O on the KV cache with  $4h_1 \cdot bls \cdot l \cdot (s \cdot n + n(n - 1)/2)$  bytes in total.

Let  $w$  denote the size of one-layer weights. The peak memory used to store the weights, activations, and KV caches can be estimated as

$$\text{peak\_mem} = w + 2h_1 \cdot bls + 4h_1 \cdot bls \cdot l \cdot (s + n)$$

If we only swap with CPU, then there is the constraint that  $\text{peak\_mem} < \text{CPU memory} - \text{some overhead}$ . Let  $cmem$  denote the right hand, there is

$$bls \leq \frac{cmem - w}{2h_1 + 4h_1 \cdot l \cdot (s + n)} = bls_1$$

Now we show that there is a better schedule that gives the same I/O efficiency but can enlarge the  $bls$  by around 2 in some cases.

**Diagonal block schedule** Figure 5 is an illustration of our diagonal block schedule. We have a block containing 4 GPU batches, and we are going to generate 4 tokens with a model that has 4 layers. There will be a one-time warm-up phase (gray area) to compute the area above the diagonal. Then for each iteration, the system will compute a diagonal that contains 4 sub-diagonals (4 squares enclosed by red outlines as the first sub-diagonal, then 4 squares enclosed by blue outlines as the second sub-diagonal). After finishing the 4 sub-diagonals, it will repeat the same computation in the next row.

For simplicity, consider the good case that the memory capacity is large enough that the diagonal can cover all  $n$  generation iterations for  $n$  tokens. The block size  $bls$  now is defined as the number of samples touched by the diagonal.

In total, to compute one diagonal, the weights of each layer will be loaded once, and the I/O of the activations and KV

cache will be in size roughly as  $1/n$  as the value in the zig-zag block schedule. There will be  $bls$  tokens generated. So the I/O per token is the same with the zig-zag block schedule after the one-time warm-up if for the same  $bls$ .

The peak memory needed to hold the necessary weights, activations, and KV cache is estimated as

$$\begin{aligned} \text{peak\_mem} &= w + 2h_1 \cdot bls \\ &+ \frac{4h_1 \cdot bls \cdot l(2s + n)(n - 1)}{2n} \end{aligned}$$

from  $\text{peak\_mem} \leq cmem$ , we have

$$bls \leq \frac{n(cmem - w)}{2h_1 \cdot n + 2h_1 \cdot l \cdot (2s + n)(n - 1)} = bls_2$$

Despite a one-time warm-up at the beginning. The diagonal block schedule can accommodate a larger block size than zig-zag block schedule at the ratio of

$$\frac{bls_2}{bls_1} = \frac{2s + 2n}{2s + n} + O\left(\frac{1}{n}\right)$$

which is close to 2 when  $n \gg s$ , and close to 1 when  $s \gg n$ .

A larger  $bls$  does not change the activations and KV caches I/O per token, but can reduce the weights I/O per token proportionally, while weights I/O can normally occupy a large portion.

**Discussions.** In offloading setting, I/O is a significant bottleneck in latency and throughput, so the diagonal block schedule should be able to give considerable gain when  $n$  is relatively large compared to  $s$  and the memory is sufficiently large to fit  $n$  samples.

When the compute resources are sufficient to avoid offloading, the diagonal block schedule can still help to reduce the peak memory and enlarge the batch size, which increases GPU utilization.

Another benefit compared to the zig-zag block schedule is that with the same throughput, the generation latency for each prompt is reduced. For example, suppose in the zig-zag block schedule the  $bls$  samples finish the generation at the same time with latency  $T$ . In the diagonal block schedule, the first  $bls/n$  samples finish the generation with latency  $T/n$ , the second  $bls/n$  samples finish with latency  $2T/n$ , and so on. The average latency of completion is reduced by half.

Despite its advantages, there are some difficulties in implementing the diagonal block schedule. The major implementation difficulty is the dynamic update of the KV cache buffer. To improve runtime efficiency, FlexGen now

pre-allocates continuous buffers for all KV cache at the beginning of a block. This works well for the zig-zag block schedule. However, for the diagonal block schedule, pre-allocating continuous buffers make it impossible to save memory anymore. To utilize the memory-saving property of the diagonal block schedule, one needs to implement efficient attention computation on non-contiguous memory.

### A.2.2. PROOF OF THEOREM 4.1

Note that in any case when we move from computing a square to another square, we need to offload and load the corresponding KV cache. So that the total I/O incurred by KV cache is constant. The total I/O incurred by activations could vary, but despite the prefill phase, its size for each square is much smaller than the KV cache for the same square. In total, the size of activations is around  $1/(2s + n)$  of the size of KV cache. We will ignore the I/O incurred by activations for simplicity, which can cause a multiplicative error of  $1/(2s + n)$  at most. Then the only thing left is the weights I/O. Starting from now, the I/O complexity in the context refers to the I/O complexity incurred by weights.

**Definition A.1.** We define the working state at any time when the GPU is computing a square as follows. Suppose there are  $k$  GPU batches working in progress. The column indices of the last squares that have been computed (including the current one) are  $a_1, a_2, \dots, a_k$ , and  $1 \leq a_i \leq n \times l$ . Different batches are identically independent, so w.l.o.g., suppose  $a_1 \geq a_2 \geq \dots \geq a_k$ . Then the working state is a tuple  $(a_1, a_2, \dots, a_k)$ . A move that does a computation on a square is a pair of states  $s^{(1)}, s^{(2)}$  that means transit from state  $s^{(1)}$  to  $s^{(2)}$ .

Consider an optimal order denoted as an infinite sequence  $m_1, m_2, \dots, m_\infty$ , where  $m_i$  is the  $i$ th move. For each  $i$ , let  $s_i$  be the current working state.

**Lemma A.2.** *If there is a list of moves that start from state  $s$ , and back to state  $s$  at the end, the number of computed squares for every column (one layer for one token) is the same.*

*Proof.* Suppose the start state  $s = (a_1, a_2, \dots, a_k)$ . For computations that occupy the whole row, the number of computed squares for every column is the same. So we only need to consider the rows that have not been fully traversed (captured by the end state). For each  $a_i$ , if the underlying row has not been finished at the end, and ends with the index  $b_i$ , then we pair  $a_i$  with  $b_i$ . If the underlying row has been finished, we pair it with a newly opened but not finished row, still, let  $b_i$  denote the new index.

Thus we have transited from state  $S_a = (a_1, a_2, \dots, a_k)$  to another state  $S_b = (b_1, b_2, \dots, b_k)$ . The indices in  $S_a$  are sorted by  $a_1 \geq a_2 \geq \dots \geq a_k$ . The indices in  $S_b$  are not sorted, but  $b_i$  is paired to  $a_i$  according to the above

paragraph. For each  $i$ , if  $b_i > a_i$ , we need to count the squares in  $(a_i, b_i]$  by 1. If  $b_i < a_i$ , we need to count the squares in  $(b_i, a_i]$  by -1. Now we argue that for each column index  $j$  and  $1 \leq j \leq n \times l$ , the count over it is summed to 0. Suppose not, that there are  $p$  positive count and  $q$  negative count and  $p \neq q$ . Then there are  $p$  values lower than  $j$  in state  $a$  and  $q$  values lower than  $j$  in state  $b$ . This contradicts the fact that  $S_a$  and  $S_b$  are the same state with different orders. Therefore, the number of computed squares for every column is the same.  $\square$

**Theorem A.3.** *The diagonal block schedule is I/O-optimal asymptotically.*

*Proof.* Notice that since the memory capacity is finite, the length of the state is finite, thus the number of the possible state is finite. If each state appears finite times in the sequence, then the sequence cannot be infinite. Therefore, there exists a state  $s$  that appears in the sequence infinite times.

Let  $j_1, j_2, \dots, j_\infty$  be the indices in the sequence that have state  $s$ . The moves between each two neighboring  $s$  states correspond to a throughput. The moves between  $j_1$  and  $j_2$  should create the highest possible throughput that pushes from state  $s$  to  $s$ . Otherwise, we can replace it to get a higher total throughput, which contradicts to that it is an optimal order. So that we can repeat such a strategy between each neighboring  $j_i, j_{i+1}$  to get an optimal compute order.

Now the problem is reduced to finding an optimal compute order between  $j_1$  and  $j_2$ . With infinite loops, the highest throughput from  $j_1$  to  $j_2$  gives the highest throughput among the whole sequence.

Assume an optimal compute order between  $j_1$  and  $j_2$ . From Lemma A.2, there is the same number of squares to be computed for every column denoted as  $c$ . With such fixed  $c$ , the throughput is determined by the I/O time between  $j_1$  and  $j_2$ . The number of times we load weights for each color in Figure 2 determines the total I/O time. Each time we load weights, for example, the weights for computing the yellow squares, we cannot compute two yellow squares in the same row without other weights swaps, because the squares between them have not been computed and require other weights.

Therefore, for one load, we can only compute squares from different rows, which means all the caches and activations corresponding to those squares need to be held (either on the CPU or on the disk). Every square corresponds to some memory consumption, for example, the squares in the range of the  $i$ -th token cost caches for  $s + i - 1$  tokens. The sum of the memory consumption of all squares is a constant denoted as  $M$ . Let  $M'$  denote the memory capacity. The number of weights loading times is at least  $\lceil M/M' \rceil$ . Let

$t_w$  denote the I/O time for loading weights for one color, the optimal throughput is at most  $c/\lceil M/M' \rceil/t_w$ .

In the diagonal block schedule, after warm-up, each time with the loaded weights, the peak memory is the sum of the memory consumption of each computed square, which is the same each time we load weights. We can set it to hit  $M'$ <sup>3</sup>. Take  $c$  number of diagonals as the repeated list of moves denoted as  $\vec{q}$ . Set the starting state to be  $s$  mentioned before,  $\vec{q}$  will restore the state to  $s$  by construction. The number of weights loading times during  $\vec{q}$  is  $\lceil M/M' \rceil$ , which meets the lower bound, and achieves the throughput upper bound  $c/\lceil M/M' \rceil/t_w$ . The warm-up phase can be ignored in the setting of an infinite sequence. In summary, the diagonal block schedule is I/O optimal asymptotically.  $\square$

The zig-zag block schedule is not optimal, as the peak memory consumption is not the same each time loading the weights. When computing the layers for the last token, the peak memory is scaled with  $s + n - 1$ , while for the first token, it is scaled with  $s$ . In order to let the former fit in  $M'$ , the latter must be smaller than  $M'$ . But the memory consumption change is linear when generating the tokens, thus the average memory consumption for each weights loading can be pushed to at least  $M'/2$ . From this, the zig-zag block schedule can achieve the throughput at least  $c/\lceil M/(M'/2) \rceil/t_w$  which is  $1/2$  of the throughput upper bound. In the infinite sequence setting, this means the zig-zag block schedule can achieve an I/O complexity that is at most  $2\times$  optimal. Therefore, we have:

**Theorem 4.1.** *The I/O complexity of the zig-zag block schedule is within  $2\times$  of the optimal solution.*

### A.3. Cost Model

In this section, we present the full cost model. Note that we use a single variable to represent constants like bandwidth and TFLOPS to simplify the formulation below. In real systems, these constants vary according to the total load. We handle such dynamics by using piece-wise functions and adding regularization terms. We carefully model the dynamics by depending only on other constants (e.g., hidden size), so the optimization problem remains a linear programming problem with respect to policy variables.

Table 6 and Table 7 give the meaning of constants used in the cost model.

The object is to maximize throughput (token/s), which is equivalent to minimizing the reciprocal (s/token). Free variables are colored blue.

<sup>3</sup>The size value is discrete, we cannot exactly hit  $M'$ , but with large enough parameters, such a gap could be set to only affect the total value by less than 1%. For example, the layer could be at the tensor level to make squares extremely fine-grained.

Var	Meaning
<i>ctog_bdw</i>	CPU to GPU bandwidth
<i>gtoc_bdw</i>	GPU to CPU bandwidth
<i>dtoc_bdw</i>	disk to CPU bandwidth
<i>ctod_bdw</i>	CPU to disk bandwidth
<i>mm_flops</i>	GPU flops per second for matrix multiplication
<i>bmm_flops</i>	GPU flops per second for batched matrix multiplication
<i>cpu_flops</i>	CPU flops per second
<i>wg</i>	percentage of weights on GPU
<i>wc</i>	percentage of weights on CPU
<i>wd</i>	percentage of weights on disk
<i>cg</i>	percentage of KV cache on GPU
<i>cc</i>	percentage of KV cache on CPU
<i>cd</i>	percentage of KV cache on disk
<i>hg</i>	percentage of activations on GPU
<i>hc</i>	percentage of activations on CPU
<i>hd</i>	percentage of activations on disk

Table 7. Notation Variables

## Objective

$$\text{Minimize } T/\text{bls}$$

Then the following constraints describe the calculation of total latency:

$$T = T_{\text{pre}} \cdot l + T_{\text{gen}} \cdot (n - 1) \cdot l$$

$$T_{\text{pre}} = \max(ctog^p, gtoc^p, dtoc^p, ctod^p, comp^p)$$

$$\begin{aligned} ctog^p &= \frac{\text{weights\_ctog}^p + \text{act\_ctog}^p}{\text{ctog\_bdw}} \\ &= \frac{1}{\text{ctog\_bdw}}((\text{wc} + \text{wd})(8h_1^2 + 4h_1 \cdot h_2) \\ &\quad + 2(\text{hc} + \text{hd})s \cdot h_1 \cdot \text{bls}) \end{aligned}$$

$$\begin{aligned} gtoc^p &= \frac{\text{cache\_gtoc}^p + \text{act\_gtoc}^p}{\text{gtoc\_bdw}} \\ &= \frac{1}{\text{gtoc\_bdw}}(4(\text{cc} + \text{cd})(s + 1)h_1 \cdot \text{bls} \\ &\quad + 2(\text{hc} + \text{hd})s \cdot h_1 \cdot \text{bls}) \end{aligned}$$

$$\begin{aligned} dtoc^p &= \frac{\text{weights\_dtoc}^p + \text{act\_dtoc}^p}{\text{dtoc\_bdw}} \\ &= \frac{1}{\text{dtoc\_bdw}}(\text{wd}(8h_1^2 + 4h_1 \cdot h_2) \\ &\quad + 2\text{hd} \cdot s \cdot h_1 \cdot \text{bls}) \end{aligned}$$

$$\begin{aligned} ctod^p &= \frac{\text{cache\_ctod}^p + \text{act\_ctod}^p}{\text{ctod\_bdw}} \\ &= \frac{1}{\text{ctod\_bdw}}(4\text{cd} \cdot \text{bls} \cdot (s + 1) \cdot h_1 \\ &\quad + 2\text{hd} \cdot s \cdot h_1 \cdot \text{bls}) \end{aligned}$$

$$\begin{aligned} comp^p &= \frac{\text{linear\_layer}^p}{\text{mm\_flops}} + \frac{\text{att}^p}{\text{bmm\_flops}} \\ &= \frac{\text{bls}(8s \cdot h_1^2 + 4s \cdot h_1 \cdot h_2)}{\text{mm\_flops}} \\ &\quad + \frac{4\text{bls} \cdot s^2 \cdot h_1}{\text{bmm\_flops}} \end{aligned}$$

$$T_{\text{gen}} = \max(ctog^g, gtoc^g, dtoc^g, ctod^g, comp^g)$$

$$\begin{aligned} ctog^g &= \frac{\text{weights\_ctog}^g + \text{act\_ctog}^g}{\text{ctog\_bdw}} \\ &= \frac{1}{\text{ctog\_bdw}}((\text{wc} + \text{wd})(8h_1^2 + 4h_1 \cdot h_2) \\ &\quad + 2(\text{hc} + \text{hd})h_1 \cdot \text{bls}) \end{aligned}$$

$$\begin{aligned} gtoc^g &= \frac{\text{act\_gtoc}^g}{\text{gtoc\_bdw}} \\ &= \frac{1}{\text{gtoc\_bdw}}(2(\text{hc} + \text{hd}) \cdot h_1 \cdot \text{bls}) \end{aligned}$$

$$\begin{aligned} dtoc^g &= \frac{\text{cache\_dtoc}^g + \text{weights\_dtoc}^g + \text{act\_dtoc}^g}{\text{dtoc\_bdw}} \\ &= \frac{1}{\text{dtoc\_bdw}}(4\text{cd} \cdot \text{bls} \cdot (s + n/2) \cdot h_1 \\ &\quad + \text{wd}(8h_1^2 + 4h_1 \cdot h_2) \\ &\quad + 2\text{hd} \cdot h_1 \cdot \text{bls}) \end{aligned}$$

$$\begin{aligned} ctod^g &= \frac{\text{cache\_ctod}^g + \text{act\_ctod}^g}{\text{ctod\_bdw}} \\ &= \frac{1}{\text{ctod\_bdw}}(4\text{cd} \cdot \text{bls} \cdot h_1 + 2\text{hd} \cdot h_1 \cdot \text{bls}) \end{aligned}$$

$$comp^g = \text{gpu\_comp}^g + \text{cpu\_comp}^g$$

$$\begin{aligned} \text{gpu\_comp}^g &= \frac{\text{linear\_layer}^g}{\text{mm\_flops}} + \frac{\text{att}^g}{\text{bmm\_flops}} \\ &= \frac{\text{bls}(8h_1^2 + 4h_1 \cdot h_2)}{\text{mm\_flops}} \\ &\quad + \frac{4\text{cg} \cdot \text{bls} \cdot (s + n/2) \cdot h_1}{\text{bmm\_flops}} \end{aligned}$$

$$\begin{aligned} \text{cpu\_comp}^g &= \frac{\text{att}^g}{\text{cpu\_flops}} \\ &= \frac{4(\text{cc} + \text{cd})\text{bls} \cdot (s + n/2) \cdot h_1}{\text{cpu\_flops}} \end{aligned}$$

## Peak Memory Constraints

- GPU peak memory constraints during prefill:  
GPU memory used to hold a fixed percentage of weights, activations, and cache is

$$\begin{aligned} gpu\_home^p &= \text{wg} \cdot (8h_1^2 + 4h_1 \cdot h_2) \cdot l \\ &\quad + \text{hg} \cdot 2s \cdot h_1 \cdot \text{bls} \\ &\quad + 4(s+n)h_1 \cdot \text{cg} \cdot \text{bls} \cdot l. \end{aligned}$$

GPU working memory (omit mask):

$$\begin{aligned} qkv^p &= \text{gbs} \cdot (2s \cdot h_1 + 3(2s \cdot h_1)) \\ &= \text{gbs} \cdot 8s \cdot h_1 \\ att_1^p &= \text{cg} \cdot \text{gbs} \cdot (2s \cdot h_1 + 2s \cdot h_1 + 2nh \cdot s^2) \\ att_2^p &= \text{cg} \cdot \text{gbs} \cdot (2nh \cdot s^2 + 2s \cdot h_1 + 2s \cdot h_1) \\ embed^p &= \text{gbs} \cdot (2s \cdot h_1 + 2s \cdot h_1) \\ &= \text{gbs} \cdot 4s \cdot h_1 \\ mlp_1^p &= \text{gbs} \cdot 2(s \cdot h_1 + s \cdot h_2) \\ &= 2\text{gbs} \cdot s(h_1 + h_2) \\ mlp_2^p &= \text{gbs} \cdot 2(s \cdot h_2 + s \cdot h_1) \\ &= 2\text{gbs} \cdot s(h_1 + h_2) \end{aligned}$$

$$\begin{aligned} gpu\_w^p &= 2(1 - \text{wg})(8h_1^2 + 4h_1 \cdot h_2) \\ &\quad + (1 - \text{hg}) \cdot 2s \cdot h_1 \cdot \text{gbs} \\ &\quad + \max(qkv, att_1, att_2, embed, mlp_1, mlp_2) \\ gpu\_peak^p &= gpu\_home^p + gpu\_w^p < gmem \end{aligned}$$

- GPU peak memory constraints after prefill:  
GPU memory used to hold a fixed percentage of weights, activations, and cache is

$$\begin{aligned} gpu\_home^g &= \text{wg} \cdot (8h_1^2 + 4h_1 \cdot h_2) \cdot l \\ &\quad + \text{hg} \cdot 2h_1 \cdot \text{bls} \\ &\quad + 4(s+n)h_1 \cdot \text{cg} \cdot \text{bls} \cdot l. \end{aligned}$$

GPU working memory (omit mask):

$$\begin{aligned} qkv^g &= \text{gbs} \cdot (2h_1 + 3(2h_1)) = 8\text{gbs} \cdot h_1 \\ att_1^g &= \text{cg} \cdot \text{gbs} \cdot (2h_1 + 2(s+n)h_1 \\ &\quad + 2nh(s+n)) \\ att_2^g &= \text{cg} \cdot \text{gbs} \cdot (2nh(s+n) + 2(s+n)h_1 \\ &\quad + 2h_1) \\ embed^g &= \text{gbs} \cdot (2h_1 + 2h_1) = 4\text{gbs} \cdot h_1 \\ mlp_1^g &= 2\text{gbs} \cdot (h_1 + h_2) \\ mlp_2^g &= 2\text{gbs} \cdot (h_2 + h_1) \end{aligned}$$

$$\begin{aligned} gpu\_w^g &= 2(1 - \text{wg})(8h_1^2 + 4h_1 \cdot h_2) \\ &\quad + (1 - \text{hg}) \cdot 2s \cdot h_1 \cdot \text{gbs} \\ &\quad + \max(qkv^g, att_1^g, att_2^g, embed^g, mlp_1^g, mlp_2^g) \\ gpu\_peak^g &= gpu\_home^g + gpu\_w^g < gmem \end{aligned}$$

- CPU peak memory constraints during prefill:  
CPU memory used to hold a fixed percentage of weights, activations, and cache is

$$\begin{aligned} cpu\_home^p &= \text{wc} \cdot (8h_1^2 + 4h_1 \cdot h_2) \cdot l \\ &\quad + \text{hc} \cdot 2s \cdot h_1 \cdot \text{bls} \\ &\quad + 4(s+n)h_1 \cdot \text{cc} \cdot \text{bls} \cdot l. \end{aligned}$$

CPU working memory:

$$\begin{aligned} cpu\_w^p &= (1 - \text{wg})(8h_1^2 + 4h_1 \cdot h_2) \\ &\quad + (1 - \text{hg}) \cdot 2s \cdot h_1 \cdot \text{gbs}. \end{aligned}$$

$$cpu\_peak^p = cpu\_home^p + cpu\_w^p < cmem$$

- CPU peak memory constraints after prefill:  
CPU memory used to hold fixed percentage of weights, activations, and cache is

$$\begin{aligned} cpu\_home^g &= \text{wc} \cdot (8h_1^2 + 4h_1 \cdot h_2) \cdot l \\ &\quad + \text{hc} \cdot 2h_1 \cdot \text{bls} \\ &\quad + 4(s+n)h_1 \cdot \text{cc} \cdot \text{bls} \cdot l. \end{aligned}$$

CPU working memory:

$$\begin{aligned} cpu\_w^g &= \text{wd}(8h_1^2 + 4h_1 \cdot h_2) \\ &\quad + 2\text{hd} \cdot 2 \cdot h_1 \cdot \text{gbs} \\ &\quad + 2\text{cd} \cdot 4(s+n)h_1 \cdot \text{gbs} \\ &\quad + 2nh \cdot (s+n) \cdot \text{gbs} \\ &\quad + 2h_1 \cdot \text{gbs}. \end{aligned}$$

$$cpu\_peak^g = cpu\_home^g + cpu\_w^g < cmem$$

- NVMe peak memory constraints:

$$\begin{aligned} nvme\_peak &= (8h_1^2 + 4h_1 \cdot h_2) \cdot \text{wd} \cdot l \\ &\quad + \text{hd} \cdot 2s \cdot h_1 \cdot \text{bls} \\ &\quad + \text{cd} \cdot 4(s+n)h_1 \cdot \text{bls} \cdot l \\ &< nmem \end{aligned}$$

## A.4. Tables and Additional Experimental Results

**Execution Breakdown** Table 8 shows the execution time breakdown for OPT-175B running on FlexGen with the setup in Table 1.

**HELM and Data Wrangling** Table 9 lists the details of HELM integration experiments. Table 10 and Table 11 shows additional results for the data wrangling task.

**Complementary Tables for Policy Details** Table 15 and Table 16 list the concrete policy setups for the results in Table 2 for prompt length 512 and 1024, from end-to-end throughput experiments. Table 19 and Table 20 list the latency and throughput for the data points in Fig. 1 which demonstrate latency-throughput tradeoff.

**Ablation Study** Table 23 list the concrete policy setups for the main ablation study result in Table 4. Table 21 and Table 22 shows some additional ablation study on policies. In Table 23, DeepSpeed chooses to store the KV cache and activations on GPU. For OPT-30B, the weights will be stored on the CPU entirely because it cannot fit in GPU. The corresponding percentage is (0, 100, 100, 0, 100, 0). The computation order of DeepSpeed is row-by-row, so the number of GPU batches in a block is 1. The GPU batch size is set to be as large as possible, which is set to 8. For OPT-175B, the weights will be stored on disk entirely according to DeepSpeed’s strategy, since it cannot be stored on CPU. The corresponding percentage is (0, 0, 100, 0, 100, 0). The number of GPU batches in a block is 1, and the GPU batch size is 2. For “No policy search”, we use different policy changes for OPT-30B and OPT-175B to demonstrate the impact of different policy dimensions. For OPT-30B, we change the percentage for weights from (20, 80) to (0, 100), and show that the throughput does not change much. For OPT-175B, we change the number of GPU batches in a block from 8 to 1 and show that the throughput degrades significantly. For “No CPU compute”, it degrades OPT-30B more than OPT-175B because the bottleneck for OPT-175B is on disk offloading. Therefore, the gain for CPU computation is small for OPT-175B. While for OPT-30B, the disk has not been used, so the gain for CPU computation is more significant.

**Different SSD Speed** To highlight the limitation and requirements of SSD speed. We tested two kinds of disk on GCP and report the generation throughput (token/s) in Table 24 (input sequence length = 512 and output sequence length = 32).

**Additional Hardware and Sequence Length** Our methods and implementations do not depend on specific hardware architectures. It can work well on different CPU architectures (e.g., Intel, AMD) and different GPU architectures (e.g., NVIDIA Ampere, NVIDIA Turing) as long as the architectures are supported by PyTorch. Some architecture (e.g. unified memory) could be more friendly to our approach. To tune the system for different architectures, we need to fit a cost model and run policy search to generate offloading policies, which can be different according to the compute capabilities, memory capacities, and memory bandwidth of different architectures. The final absolute performance will vary, but FlexGen can be easily adapted to different architectures. We did additional experiments on a different hardware setup of 24GB RTX 3090 with 125GB CPU Memory and 1TB SSD, in addition to our previous setting of 16GB T4 with 208GB CPU Memory and 1.5TB SSD, shown in Table 12. The input sequence length is set to 512 and the output sequence length is set to 32. We can see the results follow similar trends to the setup in the main paper. FlexGen outperforms other baselines significantly. Compar-

Table 8. Execution time breakdown (seconds) for OPT-175B. The prompt length is 512. (R) denotes read and (W) denotes write.

Stage	Total	Compute	Weight (R)	Cache (R)	Cache (W)
Prefill	2711	2220	768	0	261
Decoding	11315	1498	3047	7046	124

ing this 3090 setting with the T4 setting in the main paper, the performance under the 3090 setting is worse than the T4 setting for 30B and 175B. This is because CPU memory also plays a critical role when offloading is needed, making our T4 setting with larger CPU memory better.

Table 14 and Table 13 show the results for an additional prompt length 256. As all of our benchmarks in the main paper are done with output sequence length 32, so we add two additional fixed sequence lengths in Table 17 and Table 18. The numbers are generally higher in the former one because the input sequence length is smaller and the output sequence length is larger. As the throughput is defined as (number of generated tokens) / (prefill time + generation time), such a setting makes the fraction of prefill time smaller. The numbers are generally lower in the latter one because the output sequence length is smaller.

In summary, FlexGen outperforms baselines in all newly added settings. The Compression techniques used in FlexGen are helpful only for large models that need offloading. CPU memory capacity is essential for large models that need offloading.

**Batches with Various Sequence Length** We also add experiments of one realistic use case with a mixture of prompt and output lengths (HELM benchmark) in Table 25. To batch sequences of variable lengths, FlexGen simply pads all inputs to the maximum prompt length, which is a common method used in many systems. Depending on the distribution of the prompt length, the efficiency of this simple padding method varies. For example, if most sequences have similar lengths, then the batching efficiency should be very high. If some sequences are very long and some sequences are short, then FlexGen will spend a lot of time on the useless computation of padding tokens. We use two metrics: padded throughput = (number of tokens in padded prompts + number of tokens in padded outputs) / latency and actual throughput = (number of non-padding tokens in prompts + number of non-padding tokens in outputs) / latency. To better handle prompts with various lengths, one can utilize some complementary techniques from Orca(Yu et al., 2022).

**Table 9.** The setup and running time of 7 representative sub-scenarios in the HELM integration. The running time consists of dataset downloading, model initialization, generation, and metric computation. “Prompt len” denotes the input sequence length, and “Gen len” denotes the output sequence length. “Num seq” denotes the number of sequences (prompts). “time” denotes the running time in minutes.

Scenario description	Prompt len	Gen len	Num seq	time
wikifact: k=5, subject=plaintiff	256	8	288	10
wikifact: k=5, subject=instance_of	256	8	2592	55
mmlu: subject=abstract_algebra	512	1	864	31
mmlu: subject=us_foreign_policy	512	1	1008	33
synthetic_reasoning: mode=pattern_match	256	50	1584	118
synthetic_reasoning_natural: difficulty=easy	512	20	1584	100
summarization_xsum: temperature=0.3	1984	64	1568	902

**Table 10.** The setup and running time of 6 representative data wrangling tasks with OPT-30B. Because the output seq. length is short for this task, we use a new metric total throughput = (number of tokens in the prompt + number of generated tokens) / total latency.

Task	Number of seq.	Input seq. length	Output seq. length	Running time (s)	Total throughput (token/s)
EM: Fodors-Zagats	189	744	3	541.550	248.287
EM: Beer	91	592	3	238.58	224.450
EM: iTunes-Amazon	109	529	3	267.639	198.775
DI: Restaurant	86	123	5	60.310	169.790
DI: Buy	65	488	10	185.882	160.747
ED: Hospital	200	200	3	158.329	256.429

**Table 11.** The setup and running time of 6 representative data wrangling tasks with OPT-175B. Because the output seq. length is short for this task, we use a new metric total throughput = (number of tokens in the prompt + number of generated tokens) / total latency.

Task	Number of seq.	Input seq. length	Output seq. length	Running time (s)	Total throughput (token/s)
EM: Fodors-Zagats	189	744	3	3928.310	34.228
EM: Beer	91	592	3	1356.786	35.083
EM: iTunes-Amazon	109	529	3	1569.062	33.906
DI: Restaurant	86	123	5	648.762	16.968
DI: Buy	65	488	10	2086.961	14.317
ED: Hospital	200	200	3	1154.133	35.178

**Table 12.** Generation throughput (token/s) on 1 GPU (RTX 3090) with 125 GB CPU memory and 1TB SSD, run with **input sequence length 512 and output sequence length 32**. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. The gray tuple denotes a policy (GPU batch size  $\times$  #GPU-batch,  $wg, wc, cg, cc, hg, hc$ ).

Seq. length	512 + 32			
	Model size	6.7B	30B	175B
Accelerate	183.177 (16×1, 100, 0, 100, 0, 100, 0)	2.077 (13×1, 0, 100, 100, 0, 100, 0)	0.026 (4×1, 0, 0, 100, 0, 100, 0)	
DeepSpeed	38.027 (32×1, 0, 100, 100, 0, 100, 0)	3.889 (12×1, 0, 100, 100, 0, 100, 0)	0.019 (3×1, 0, 0, 100, 0, 100, 0)	
FlexGen	233.756 (28×1, 100, 0, 100, 0, 100, 0)	5.726 (4×15, 25, 75, 40, 60, 100, 0)	0.384 (64×4, 0, 25, 0, 0, 100, 0)	
FlexGen (c)	120.178 (144×1, 100, 0, 100, 0, 100, 0)	16.547 (96×2, 25, 75, 0, 100, 100, 0)	1.114 (24×1, 0, 100, 0, 100, 100, 0)	

**Table 13.** Generation throughput (token/s) on 1 GPU with different systems. Accelerate, DeepSpeed, and FlexGen use 1 GPU. Petals uses 1 GPU for OPT-6.7B, 4 GPUs for OPT-30B, and 24 GPUs for OPT-175B, but reports per-GPU throughput. Petals is benchmarked under different network delay and bandwidth. The models are run in INT8 as the default for Petals. We tune the batch size of each request to be 2 and issue requests by 6 parallel client processes to achieve the maximum throughput. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. “OOM” means out-of-memory.

Seq. length	256			512			1024		
Model size	6.7B	30B	175B	6.7B	30B	175B	6.7B	30B	175B
Accelerate	50.66	1.34	0.02	25.12	0.62	0.01	13.01	0.31	0.01
DeepSpeed	14.52	1.30	0.01	9.28	0.60	0.01	4.59	0.29	OOM
Petals (<5ms, 1Gb/s)	9.03	3.55	0.09	8.25	2.84	0.08	6.56	1.51	0.06
Petals (<5ms, 100Mb/s)	9.15	2.53	0.06	8.18	1.67	0.05	6.52	0.87	0.03
Petals (100ms, 100Mb/s)	8.64	0.75	0.01	7.82	0.64	0.01	5.89	0.37	0.01
FlexGen	53.29	16.01	1.36	25.26	7.32	0.69	13.72	3.50	0.35
FlexGen (c)	56.72	16.86	2.26	29.12	8.70	1.12	13.18	3.98	0.42

**Table 14.** Generation throughput (token/s) on 1 GPU with **input sequence length 256 and output sequence length 32**. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. “OOM” means out-of-memory. The gray tuple denotes a policy (GPU batch size  $\times$  #GPU-batch,  $w_g, w_c, c_g, c_c, h_g, h_c$ ).

Seq. length	256		
Model size	6.7B	30B	175B
Accelerate	50.66 (4×1, 100, 0, 100, 0, 100, 0)	1.34 (16×1, 0, 100, 100, 0, 100, 0)	0.02 (4×1, 0, 0, 100, 0, 100, 0)
DeepSpeed	14.52 (32×1, 0, 100, 100, 0, 100, 0)	1.30 (12×1, 0, 100, 100, 0, 100, 0)	0.01 (2×1, 0, 0, 100, 0, 100, 0)
FlexGen	53.29 (4×1, 100, 0, 100, 0, 100, 0)	16.01 (160×2, 10, 90, 0, 100, 0, 100)	1.36 (64×8, 0, 50, 0, 0, 0, 100)
FlexGen (c)	56.72 (128×1, 100, 0, 100, 0, 100, 0)	16.86 (128×8, 0, 100, 0, 100, 0, 100)	2.26 (96×3, 0, 100, 0, 100, 0, 100)

**Table 15.** Generation throughput (token/s) on 1 T4 GPU with **input sequence length 512 and output sequence length 32**. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. “OOM” means out-of-memory. The gray tuple denotes a policy (GPU batch size  $\times$  #GPU-batch,  $w_g, w_c, c_g, c_c, h_g, h_c$ ).

Seq. length	512		
Model size	6.7B	30B	175B
Accelerate	25.12 (2×1, 100, 0, 100, 0, 100, 0)	0.62 (8×1, 0, 100, 100, 0, 100, 0)	0.01 (2×1, 0, 0, 100, 0, 100, 0)
DeepSpeed	9.28 (16×1, 0, 100, 100, 0, 100, 0)	0.60 (4×1, 0, 100, 100, 0, 100, 0)	0.01 (1×1, 0, 0, 100, 0, 100, 0)
FlexGen	25.26 (2×1, 100, 0, 100, 0, 100, 0)	7.32 (48×3, 20, 80, 0, 100, 0, 100)	0.69 (32×8, 0, 50, 0, 0, 0, 100)
FlexGen (c)	29.12 (72×1, 100, 0, 100, 0, 100, 0)	8.70 (16×20, 20, 80, 0, 100, 0, 100)	1.12 (48×3, 0, 100, 0, 100, 0, 100)

**Table 16.** Generation throughput (token/s) on 1 T4 GPU with **input sequence length 1024 and output sequence length 32**. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. “OOM” means out-of-memory. The gray tuple denotes a policy (GPU batch size  $\times$  #GPU-batch,  $w_g, w_c, c_g, c_c, h_g, h_c$ ).

Seq. length	1024		
Model size	6.7B	30B	175B
Accelerate	13.01 (1×1, 100, 0, 100, 0, 100, 0)	0.31 (4×1, 0, 100, 100, 0, 100, 0)	0.01 (1×1, 0, 0, 100, 0, 100, 0)
DeepSpeed	4.59 (8×1, 0, 100, 100, 0, 100, 0)	0.29 (2×1, 0, 100, 100, 0, 100, 0)	OOM
FlexGen	13.72 (1×1, 100, 0, 100, 0, 100, 0)	3.50 (20×4, 4, 96, 0, 100, 0, 100)	0.35 (12×12, 0, 50, 0, 0, 0, 100)
FlexGen (c)	13.18 (28×1, 100, 0, 100, 0, 100, 0)	3.98 (20×12, 0, 100, 0, 100, 0, 100)	0.42 (12×4, 0, 100, 0, 100, 0, 100)

**Table 17.** Generation throughput (token/s) on 1 T4 GPU with **input sequence length 128 and output sequence length 128**. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. “OOM” means out-of-memory. The gray tuple denotes a policy (GPU batch size  $\times$  #GPU-batch,  $wg, wc, cg, cc, hg, hc$ ).

Seq. length	128 + 128		
Model size	6.7B	30B	175B
Accelerate	73.411 (5 $\times$ 1, 100, 0, 100, 0, 100, 0)	1.547 (16 $\times$ 1, 0, 100, 100, 0, 100, 0)	0.021 (4 $\times$ 1, 0, 0, 100, 0, 100, 0)
DeepSpeed	19.193 (36 $\times$ 1, 0, 100, 100, 0, 100, 0)	1.717 (12 $\times$ 1, 0, 100, 100, 0, 100, 0)	0.024 (3 $\times$ 1, 0, 0, 100, 0, 100, 0)
FlexGen	106.404 (7 $\times$ 1, 100, 0, 100, 0, 100, 0)	24.634 (32 $\times$ 10, 25, 75, 0, 100, 100, 0)	2.409 (64 $\times$ 8, 0, 50, 0, 0, 0, 100)
FlexGen (c)	92.568 (196 $\times$ 1, 100, 0, 100, 0, 100, 0)	39.141 (128 $\times$ 8, 25, 75, 0, 100, 0, 100)	4.264 (80 $\times$ 3, 0, 100, 0, 100, 100, 0)

**Table 18.** Generation throughput (token/s) on 1 T4 GPU with **input sequence length 512 and output sequence length 8**. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. “OOM” means out-of-memory. The gray tuple denotes a policy (GPU batch size  $\times$  #GPU-batch,  $wg, wc, cg, cc, hg, hc$ ).

Seq. length	512 + 8		
Model size	6.7B	30B	175B
Accelerate	17.290 (2 $\times$ 1, 100, 0, 100, 0, 100, 0)	0.628 (7 $\times$ 1, 0, 100, 100, 0, 100, 0)	0.009 (2 $\times$ 1, 0, 0, 100, 0, 100, 0)
DeepSpeed	9.055 (18 $\times$ 1, 0, 100, 100, 0, 100, 0)	0.872 (6 $\times$ 1, 0, 100, 100, 0, 100, 0)	0.007 (1 $\times$ 1, 0, 0, 100, 0, 100, 0)
FlexGen	16.425 (2 $\times$ 1, 100, 0, 100, 0, 100, 0)	3.938 (512 $\times$ 8, 20, 80, 0, 100, 0, 100)	0.451 (32 $\times$ 8, 0, 50, 0, 0, 0, 100)
FlexGen (c)	14.244 (76 $\times$ 1, 100, 0, 100, 0, 100, 0)	4.019 (16 $\times$ 36, 25, 75, 0, 100, 0, 100)	0.559 (48 $\times$ 3, 0, 100, 0, 100, 0, 100)

**Table 19.** The Pareto frontier of the latency-throughput trade-off of OPT-175B. The numbers are generation throughput (token/s) and effective batch latency (s) on 1 GPU with **input sequence length 512 and output sequence length 32**. The numbers in the parentheses are corresponding effective batch sizes. The numbers in bold are the best throughput and latency for each model. We organize the table so that the latency numbers of different methods in each row are similar for each model. The top value of each column corresponds to the setting of effective batch size 1. (To reach the lowest latency, FlexGen uses an effective batch size of 2 rather than 1 because the latency difference between batch sizes 1 and 2 is negligible in this case. So, a run with batch size 2 dominates the one with batch size 1 with higher throughput and similar latency.)

175B (generation throughput / latency)				
Accelerate	DeepSpeed	FlexGen	FlexGen (c)	
-	-	-	0.052 / <b>612</b> (1)	
-	-	-	0.198 / 647 (4)	
-	-	-	0.369 / 693 (8)	
-	-	-	0.779 / 1973 (48)	
-	-	0.025 / 2555 (2)	1.092 / 2813 (96)	
-	-	0.254 / 4028 (32)	<b>1.122</b> / 4072 (144)	
-	0.006 / 5024 (1)	0.421 / 4864 (64)	-	
-	-	0.572 / 7159 (128)	-	
0.004 / 7508 (1)	-	-	-	
0.008 / 7633 (2)	-	-	-	
-	-	0.687 / 11916 (256)	-	

**Table 20.** The Pareto frontier of the latency-throughput trade-off of OPT-30B. The numbers are generation throughput (token/s) and effective batch latency (s) on 1 GPU with **input sequence length 512 and output sequence length 32**. The numbers in the parentheses are corresponding effective batch sizes. The numbers in bold are the best throughput and latency for each model. We organize the table so that the latency numbers of different methods in each row are similar for each model. The top value of each column corresponds to the setting of effective batch size 1.

30B (generation throughput / latency)				
Accelerate	DeepSpeed	FlexGen	FlexGen (c)	
-	-	-	0.21 / <b>153</b> (1)	
-	-	-	0.42 / 154 (2)	
-	-	0.20 / 159 (1)	0.82 / 155 (4)	
-	-	0.37 / 172 (2)	1.58 / 162 (8)	
-	-	0.73 / 174 (4)	2.88 / 178 (16)	
-	0.16 / 203 (1)	1.40 / 183 (8)	-	
-	0.31 / 204 (2)	2.70 / 190 (16)	-	
-	0.62 / 206 (4)	4.05 / 253 (32)	4.63 / 277 (40)	
0.08 / 405 (1)	-	5.71 / 359 (64)	6.72 / 381 (80)	
0.31 / 408 (4)	-	-	-	
0.62 / 413 (8)	-	-	-	
-	-	7.32 / 559 (144)	-	
-	-	-	7.96 / 644 (160)	
-	-	-	8.49 / 904 (240)	
-	-	-	<b>8.70</b> / 1177 (320)	

**Table 21.** Ablation study of policies. The numbers correspond to generation **throughput** on 1 GPU with **input sequence length 512 and output sequence length 32**. All policies have CPU computation turned on. The numbers for OPT-175B show some inconsistency with the end-to-end evaluation in Table 2 and Table 15 (0.49 vs 0.69) because we turn on the pagecache-management (Morton, 2008) tool to prevent the automatic disk cache in operating systems, which makes the ablation results more accurate but brings some overheads. This added some overhead and misses the advantage of using CPU cache. A real run should be expected to have a better throughput. (*gbs* denotes the GPU batch size, *#gb* denotes the number of GPU batches in a block.)

<i>gbs</i>	<i>#gb</i>	<i>wg</i>	<i>wc</i>	<i>cg</i>	<i>cc</i>	<i>hg</i>	<i>hc</i>	30B (token/s)	175B (token/s)
48	3	20	80	0	100	0	100	<b>7.32</b>	OOM
48	3	0	100	0	100	0	100	7.26	OOM
48	1	20	80	0	100	0	100	5.40	OOM
32	8	0	50	0	0	0	100	1.66	<b>0.49</b>
32	8	0	0	0	0	0	100	1.55	0.44
32	1	0	50	0	0	0	100	0.88	0.23
1	1	20	80	100	0	100	0	0.20	OOM
1	1	0	50	100	0	100	0	0.04	0.01
8	1	0	100	100	0	100	0	1.57	OOM
2	1	0	0	100	0	100	0	0.05	0.01

**Table 22.** Ablation study of policies. The numbers are full generation **latency** on 1 GPU with **input sequence length 512 and output sequence length 32**. All policies have CPU computation turned on. We turn on the pagecache-mangagement (Morton, 2008) tool to prevent the automatic disk cache in operating systems, which makes the ablation results more accurate but brings some overheads. This added some overhead and misses the advantage of using CPU cache. A real run should be expected to have a better latency. ( $gbs$  denotes the GPU batch size,  $\#gb$  denotes the number of GPU batches in a block.)

$gbs$	$\#gb$	$wg$	$wc$	$cg$	$cc$	$hg$	$hc$	30B (s)	175B (s)
48	3	20	80	0	100	0	100	559	OOM
48	3	0	100	0	100	0	100	635	OOM
48	1	20	80	0	100	0	100	284	OOM
32	8	0	50	0	0	0	100	4930	16611
32	8	0	0	0	0	0	100	5287	18704
32	1	0	50	0	0	0	100	1164	4476
1	1	20	80	100	0	100	0	<b>160</b>	OOM
1	1	0	50	100	0	100	0	737	<b>3107</b>
8	1	0	100	100	0	100	0	170	OOM
2	1	0	0	100	0	100	0	1215	6072

**Table 23.** Ablation study of proposed techniques. The numbers are generation throughput on 1 T4 GPU with prompt length 512 and generating length 32. The gray tuple denotes a policy (GPU batch size  $\times$  #GPU-batch,  $wg, wc, cg, cc, hg, hc$ ).

Model size	30B	175B
All optimizations	7.32 (48 $\times$ 3, 20, 80, 0, 100, 0, 100)	0.69 (32 $\times$ 8, 0, 50, 0, 0, 0, 100)
No policy search	7.26 (48 $\times$ 3, 0, 100, 0, 100, 0, 100)	0.27 (32 $\times$ 1, 0, 50, 0, 0, 0, 100)
No overlapping	5.86 (48 $\times$ 3, 20, 80, 0, 100, 0, 100)	0.59 (32 $\times$ 8, 0, 50, 0, 0, 0, 100)
No CPU compute	4.03 (48 $\times$ 3, 20, 80, 0, 100, 0, 100)	0.62 (32 $\times$ 8, 0, 50, 0, 0, 0, 100)
No disk	7.32 (48 $\times$ 3, 20, 80, 0, 100, 0, 100)	OOM
w/ DeepSpeed policy	1.57 (8 $\times$ 1, 0, 100, 100, 0, 100, 0)	0.01 (2 $\times$ 1, 0, 0, 100, 0, 100, 0)

**Table 24.** Generation throughput (token/s) on hardware specified in Table 1 with **input sequence length 512 and output sequence length 32**. The performance of OPT-30B is not affected because OPT-30B does not use SSD. The disk speed is measured using the Linux command dd with a block size (bs) of 1MB and the number of blocks (count) of 16000. The PageCacheManagement tool is used to disable disk cache in the operating system during measurement.

Disk Specification	30B	175B
1.6GB/s read, 1.3GB/s write (local SSD, the one used in the main paper)	7.32	0.69
0.5GB/s read, 0.5GB/s write (persistent SSD, a new setting)	7.32	0.30
1.6GB/s read, 1.3GB/s write (local SSD, use PageCacheManagement)	7.32	0.49
0.5GB/s read, 0.5GB/s write (persistent SSD, use PageCacheManagement)	7.32	0.292

**Table 25.** Selected example of FlexGen on real-world tasks from the HELM benchmark, which consists of prompts of various lengths with different output lengths. We use two metrics: padded throughput = (number of tokens in padded prompts + number of tokens in padded outputs) / latency, actual throughput = (number of non-padding tokens in prompts + number of non-padding tokens in outputs) / latency. The throughput are measured in token/s. To batch sequences of variable lengths, FlexGen simply pads all inputs to the maximum prompt length, which is a common method used in many systems. Depending on the distribution of the prompt length, the efficiency of this simple padding method varies. For example, if most sequences have similar lengths, then the batching efficiency should be very high. if some sequences are very long and some sequences are short, then FlexGen will spend a lot of time on the useless computation of padding tokens.

Task	Padded input seq. length	Padded output seq. length	Padded throughput	Actual throughput	Efficiency
MMLU (abstract_algebra)	512	1	251.5	188.6	75.0%
xsum	1984	64	60.5	47.6	78.7%

# Flexible Performant GEMM Kernels on GPUs

Thomas Faingnaert, Tim Besard, Bjorn De Sutter, *Member, IEEE*

**Abstract**—General Matrix Multiplication or GEMM kernels take centre place in high performance computing and machine learning. Recent NVIDIA GPUs include GEMM accelerators, such as NVIDIA’s Tensor Cores. Their exploitation is hampered by the two-language problem: it requires either low-level programming which implies low programmer productivity or using libraries that only offer a limited set of components. Because rephrasing algorithms in terms of established components often introduces overhead, the libraries’ lack of flexibility limits the freedom to explore new algorithms. Researchers using GEMMs can hence not enjoy programming productivity, high performance, and research flexibility at once. In this paper we solve this problem. We present three sets of abstractions and interfaces to program GEMMs within the scientific Julia programming language. The interfaces and abstractions are co-designed for researchers’ needs and Julia’s features to achieve sufficient separation of concerns and flexibility to easily extend basic GEMMs in many different ways without paying a performance price. Comparing our GEMMs to state-of-the-art libraries CUBLAS and CUTLASS, we demonstrate that our performance is in the same ballpark of the libraries, and in some cases even exceeds it, without having to write a single line of code in CUDA C++ or assembly, and without facing flexibility limitations.

**Index Terms**—matrix multiplication, graphics processors, high-level programming languages

## 1 INTRODUCTION

GEMM (General Matrix Multiplication) kernels form the core of many computations in the fields of HPC (High Performance Computing) and ML (Machine Learning). In HPC, GEMM is at the core of linear algebra [1], including dense linear algebra [2], [3], and is used for earthquake simulation [4], plasma visualisation [5], and weather and climate prediction [6]. In ML they are used to train neural networks including fully connected layers in traditional neural networks, convolutional neural networks, long short-term memory cells, and natural language processing [7], [8]. To accelerate their computations, researchers in the mentioned domains have relied on the massively parallel computing resources of GPUs (Graphics Processing Units).

To answer the demand for more efficient GEMMs, recent GPUs include matrix multiplication accelerators, such as NVIDIA’s TCs (Tensor Cores) [9]. Researchers can exploit these resources in two ways. They can express their algorithms in high-level PLs (Programming Languages) such as Python and express them in terms of established GEMM variants for which efficient implementations are available in third-party libraries such as CUBLAS or CUTLASS. This approach offers high research productivity, at the cost of being limited to the APIs (Application Programming Interfaces) and GEMM implementations available in the libraries. In many domains, this lack of flexibility is problematic. When non-standard, more generalised GEMMs as needed in neural networks [10], convolutional networks [8], fluid dynamics [11], electromechanics [12], computational chemistry [13], or any other computation on multidimensional tensors [14], [15], [16], [17], [18], [19], [20], [21], [22], [23] are rephrased

in terms of standard GEMM kernels available in libraries, additional custom kernels need to be launched in between the GEMM kernels for things such as precision conversions, layout conversions (transpositions), type conversions, bias operations, element-wise operations, etc. These extra kernels introduce huge overheads because they have a massive impact on the traffic to the very slow global memory.

Alternatively, researchers can rewrite the most demanding parts of their software in lower-level PLs such as CUDA C/C++ [24] or OpenCL [25]. This decreases their productivity, however, and they now require much more PL and GPU programming model knowledge outside their own application domain. In short, many researchers working with GEMM-like algorithms suffer from the *two-language problem*. They cannot achieve high performance, high research productivity, and algorithmic flexibility together.

The scientific PL Julia is designed to overcome the two-language problem [26]. It offers a high-level syntax, dynamic typing, managed memory, meta-programming, multiple dispatch, and other features that increase programmer productivity. Julia’s compiler is based on type inference and just-ahead-of-time compilation, which allows it to generate code devoid of much of the run-time overhead (e.g. in the form of dynamic type checks) that other PLs pay for supporting the mentioned features. On CPUs (Central Processing Units), Julia code is comparable in performance to C, C++, and Fortran code [27]. Through the CUDA.JL package, it is possible to program NVIDIA GPUs directly in Julia, at a high abstraction level of arrays or at the lower-level of CUDA-like kernels [28], [29]. Before our research, TCs were not supported in CUDA.JL, however. The package and its high-level APIs were hence of limited use to many researchers.

In our research, we set out to overcome this issue in three steps. First, we developed support for TCs in the Julia compiler and libraries through a WMMA API of wrapper functions around so-called compiler intrinsics. This allows for exploiting TCs in kernels written with the lower-level support in CUDA.JL. This low-level API only focuses on the

• T. Faingnaert and B. De Sutter are with the Department of Electronics and Information Systems, Ghent University, Belgium. T. Besard works for Julia Computing.

thomas.faingnaert@ugent.be; tim@juliacomputing.com

Corresponding author: bjorn.desutter@ugent.be

Manuscript received X.; revised X.

WMMA operation. It does not free the programmer from the cumbersome task of coordinating the memory traffic in the memory hierarchy to move data to and from the TCs. So secondly, we developed a tiling API in Julia that allows programmers to coordinate the memory traffic to and from TCs at a high abstraction level, and, importantly, without paying a price in terms of performance. The low-level API and the tiling API enable efficient use of TCs and the GPU memory hierarchy, but to reach good performance, the GEMM computations themselves, possibly with fused additional computations, then still need to be programmed at a rather low level of abstraction requiring a lot of expertise. In the final step, we therefore developed a high-level GEMM API in Julia that allows programmers to express and combine a range of extensions of basic GEMM computations in an abstract, intuitive way, without having to pay an unacceptable price in performance. Combined, these three APIs solve the two-language problem to a great extent with respect to hardware resources such as TCs.

Our main result is that we get performance in the same ballpark of hand-tuned libraries and in some cases even much better performance, without having to write a single line of code in a lower-level PL and without being limited to the specific GEMM versions supported by the libraries.

This paper focuses on the tiling and GEMM APIs. After providing the necessary background in Section 2, Section 3 discusses requirements for a tiling API, presents our novel way for abstracting tiling and the Julia API we designed based on that abstraction, and demonstrates and evaluates the API on a number of stages in GEMM computations. Section 4 discusses the requirements for flexibility in GEMMs in more detail. We present the different building blocks at the basis of our Julia GEMM API that provide that flexibility in an intuitive manner, and we demonstrate the API on a number of examples. In Section 5, our final contribution is a performance evaluation of multiple variants of GEMM computations, showing that we get relatively close to the performance of hand-tuned libraries like CUBLAS, CUTLASS, and CUTENSOR without having to write any single line in a lower-level PL. The paper then ends with a discussion of some related work in Section 6, the availability of our artefacts in Section 7, and with a conclusion and a look forward in Section 8.

## 2 BACKGROUND

### 2.1 GPU programming

The main difference between programming GPUs versus CPUs is their underlying programming model. GPUs are massively parallel processors, meaning that a large number of threads execute the same function in parallel. In GPU parlance, this function is commonly referred to as a *kernel*.

GPU threads are organised in a thread hierarchy [24]. Since our main interest is in NVIDIA GPUs, we limit our discussion to NVIDIA's CUDA programming model. *Threads* are the smallest unit of execution in the hierarchy. The hardware groups them into sets of 32 threads called *warps*. Threads in the same warp execute in a SIMT (Single Instruction Multiple Thread) fashion. These threads must hence execute the same instruction at the same time, possibly on different data. Threads are also grouped by the programmer into *blocks*.

Threads in the same block can communicate efficiently, so that they can cooperate on a common task. Finally, the set of all blocks on the GPU device is called the *grid*.

Similarly to threads, GPU memory is also ordered hierarchically. We are mainly interested in three parts of this hierarchy, which correspond directly to levels in the thread hierarchy. The *register* file is the fastest type of memory. Each thread typically has access to 255 registers. Each block has its own set of *shared memory*, that may be used by threads in the same block to communicate. Finally, *global memory* can be accessed by all threads on the device, regardless of which block they belong to. Global memory has the largest capacity, but also has much higher latency and lower throughput.

To fully exploit the available resources on a GPU, programmers can either use low-level PLs like CUDA C/C++ [24] or OpenCL [25] to program their own kernels, or they can use the foreign function interface of high-level PLs such as Python to invoke kernels in libraries. The former option requires quite some knowledge in GPU programming models, forces the programmers to write quite some boilerplate code to manage data in memories and configure the kernels, and offers little performance portability, so manual (re)tuning of code is necessary when porting the code to different devices. Popular libraries such as CUBLAS contain kernel versions tuned for many different devices to overcome the performance portability issue.

### 2.2 Julia Programming Language

The open-source PL Julia features a high-level syntax [30]. A central paradigm in its design is the way it handles dispatch, the process by which the compiler chooses which implementation of a function to use for a given function call. Julia uses a *multiple dispatch* scheme, which means that this choice depends on the types of *all* of a function's arguments.

Julia's type system is *dynamic*, meaning that the types of expressions are not necessarily known statically. However, Julia inherits some of the advantages of static type systems through several features of its compiler. For one, the Julia compiler applies type inference to deduce the types of values used by the program. Code is then *specialised* based on this information, e.g. function calls are devirtualised, dynamic type checks are removed, etc. This style of compilation, dubbed *just-ahead-of-time*, has the performance of ahead-of-time compiled PLs with the flexibility of a just-in-time compiled one. We rely on this design to seamlessly compose a GEMM computation from all involved components, i.e. beyond what normal layering of libraries at different layers of abstraction allows as is typically done with other PLs.

Julia's compiler is built on top of LLVM, a compiler infrastructure project commonly used in research and industry [31]. Julia's compilation process consists of a couple steps. First, Julia code is converted to an IR (Intermediate Representation) that is used for type inference, called Julia IR. Next, Julia IR is lowered to LLVM IR, the representation that LLVM uses. From this point onwards, the LLVM framework takes control of the compilation process. LLVM contains a set of backends, one for each target architecture that LLVM supports. The backend corresponding to the current architecture will then convert this LLVM IR to native instructions.

The Julia package CUDA.JL reuses part of the Julia compilation process to allow executing kernels written in

Julia on NVIDIA GPUs [28]. In particular, the aforementioned compilation pipeline is run to the point where Julia IR is lowered to LLVM IR. The LLVM IR is intercepted and sent to the LLVM NVPTX backend instead of the backend of the host architecture. This NVPTX backend converts the IR to PTX (Parallel Thread Execution) instructions, the virtual instruction set of NVIDIA GPUs.

With CUDA.JL, it is possible to program NVIDIA GPUs at the lower-level of CUDA-like kernels and at the higher abstraction level of arrays [28]. The former involves less boilerplate and verbosity, and makes reusing code easier compared to programming in CUDA C/C++. The latter enables much more productive programming [29].

Julia’s multiple dispatch enables transparent exploitation of performance-optimised functionality from popular GPU libraries. For example, for any function from the CUBLAS library, a Julia package can contain a generic implementation in pure Julia code that operates for all (numeric) data types and that hence accepts all arguments of type `Number`. In addition, the package can contain wrappers that each only accept a more concrete argument type such as `Float32` and that invoke the corresponding CUBLAS function for that type. Users of the package can then invoke the function on any type they want. If it is supported by the CUBLAS library, they will get optimal performance “for free”.

### 2.3 Tensor Cores

Each TC performs a matrix multiply-accumulate expression of the form  $D = A \cdot B + C$ . TCs support a limited set of possible data types for these matrices. For example, if the  $A$  and  $B$  matrices are stored as 16-bit floating point values, the  $C$  and  $D$  matrices are 32-bit floating point.

NVIDIA exposes TCs in C++ in the so-called WMMA (Warp Matrix Multiply Accumulate) API. WMMA instructions must be used by all threads in a warp in a SIMT fashion. Each thread that cooperates in a warp-wide WMMA operation holds a part of each matrix in its registers, called a *fragment*. In the remainder of this paper, unless stated differently, we will assume that  $A$  is an  $M \times K$  matrix,  $B$  is a  $K \times N$  matrix, and  $C$  and  $D$  are  $M \times N$  matrices. The tuple  $(M, N, K)$  is called the *shape* of the WMMA operation. Not all possible values of  $M$ ,  $N$ , and  $K$  are allowed, as WMMA restricts the set of possible shapes. Conceptually, WMMA consists of three separate steps:

- 1) Load the input matrices  $A$ ,  $B$ , and  $C$  from memory into WMMA fragments using a WMMA LOAD operation.
- 2) Perform the matrix multiply-accumulate using a WMMA MMA operation, resulting in a fragment of  $D$ .
- 3) Store the resultant  $D$  fragment to memory using a WMMA STORE operation.

In CUDA C++ each step corresponds to an overloaded C++ function. Calls to these functions are mapped one-to-one onto the corresponding WMMA PTX instruction by the compiler.

To add support for WMMA to CUDA.JL, we reused the pre-existing WMMA PTX intrinsics in the NVPTX backend. This necessitated adaptations to Julia’s compiler, in particular to the code generation process. Our WMMA API consists of two different layers. The lowest layer consists of Julia wrapper functions that are mapped one-to-one to these intrinsics. The second layer is a high-level interface, similar

to CUDA C++’s version of WMMA. It consists of `load_a`, `load_b`, `load_c`, `mma`, and `store_d` functions, which call the intrinsic wrapper corresponding to the argument types.

At its launch with the Volta architecture in 2017, WMMA only supported  $16 \times 16 \times 16$  multiply-accumulates of FP16 matrices. More recent GPU architectures extend the interface with new data types and shapes. Turing’s second generation TCs, introduced in 2018, add support for 8-bit, 4-bit, and 1-bit data types, along with new WMMA shapes depending on the data type used. The most recent version of WMMA includes support for FP64, bfloat16, and TF32 data types, and was launched in May 2020 with the introduction of Ampere.

TCs can also be used through libraries instead of WMMA. NVIDIA’s CUDNN library contains TC kernels for common ML algorithms. ML frameworks such as TensorFlow, PyTorch, and MXNet use CUDNN for training and inference. CUBLAS, CUBLASLT, and CUTLASS contain optimised GEMM kernels for HPC applications. NVIDIA’s CUTENSOR builds on CUTLASS and contains Tensor-Core-accelerated kernels for tensor computations.

## 3 ABSTRACTIONS FOR RECURSIVE BLOCKING

### 3.1 Requirements

Matrix multiplication is rich in data reuse. For example, multiplying square matrices of size  $N$  requires  $\mathcal{O}(N^3)$  floating point operations, but only  $\mathcal{O}(N^2)$  storage, so each element is reused roughly  $\mathcal{O}(N)$  times. To exploit this reuse, data needs to be re-accessed as much as possible in faster memories. When all data does not fit into the fastest memories, the transfers between different memories in the hierarchy need to be coordinated carefully to maximise reuse.

For GEMMs, the general idea is to copy tiles of the input matrices up into the memory hierarchy: from global memory to shared memory and from there to registers. The size of the tiles in each step is chosen such that they fit in the available memory. As computations of different tiles of the resultant matrix are independent, those can be performed completely in parallel to maximise the resource utilisation of massively parallel GPUs. Because of the one-to-one mapping between levels of threads and the memory hierarchy, each of the tiled copy operations is also performed cooperatively, by all threads in the relevant part of the thread hierarchy.

Consider again the GEMM of  $D = A \cdot B + C$ . With tiling, this GEMM will consist of the following stages:

- 1) Copy a tile of  $C$  from global memory to shared memory, cooperatively by all threads in a block.
- 2) Copy a tile of  $C$  from shared memory to registers, cooperatively by all threads in a warp.
- 3) Iterate over dimension  $K$ , stride = block tiling size.
  - a) Copy a tile of  $A$  from global memory to shared memory, cooperatively by all threads in a block.
  - b) Do the same for a tile of  $B$ .
  - c) Iterate over dimension  $K$ , stride = warp tiling size.
    - i) Copy a tile of  $A$  from shared memory to registers, cooperatively by all threads in a warp.
    - ii) Do the same for a tile of  $B$ .
    - iii) Compute a tile of  $D$ , given the  $A$ ,  $B$ , and  $C$  tiles, cooperatively by all threads in a warp.

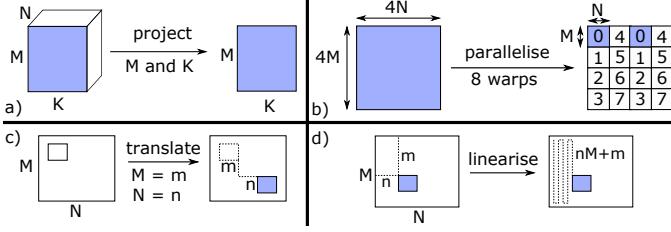


Figure 1. Projection, parallelisation, translation and linearisation of tiles.

- 4) Copy a tile of  $D$  from registers to shared memory cooperatively by all threads in a warp.
- 5) Copy a tile of  $D$  from shared memory to global memory, cooperatively by all threads in a block.

For a WMMA GEMM, stages 2, 3.c.i, and 3.c.ii correspond to WMMA LOAD operations, stage 3.c.iii to MMA operations, and stage 4 to WMMA STORE operations.

This form of recursive blocking is an absolute requirement to achieve good performance, but it is also complex to program. Tile sizes need to be chosen in function of the hardware and the number of dimensions and the sizes of the data, indexing of the matrices depends on their layouts, optimal tiling parameters can differ between memory and computational stages. Determining the tiling parameters is complex, and encoding all address computations in the actual code is cumbersome, error-prone, and results in code that is hard to comprehend, port, and maintain. To make it easier to program general GEMM computations with recursive blocking, we developed a novel API with which the tiling computations can be abstracted to a much higher level. The requirements we put forward for this API are the following:

- *Code readability* to ease writing kernels that use blocking.
- *Zero performance cost* compared to manually expressed address computations of all tiles.
- *Support for multiple dimensions (>2)* to support TCTs (Tensor ConContractions) and batched GEMMs.
- *Recursive blocking* for the different levels of the memory hierarchy through independent tiling parameters.
- *WMMA-compatibility* to exploit WMMA.

### 3.2 Abstract Operations

To meet the requirements, we propose a novel abstraction consisting of four different operations on tiles.

*Projection* is the first abstraction. The compute stages of GEMM will use tiles that refer to the three-dimensional iteration space  $(M, N, K)$ . In the memory stages of GEMM, we typically only need two of these dimensions. For example, to load a slice of the  $A$  matrix, we are only interested in the  $M$  and  $K$  dimension. Projecting a tile reduces its dimensionality by dropping one or more of its dimensions, as shown in Figure 1a. The projection abstraction thus allows us to easily reduce the original three-dimensional tile to a tile containing only the relevant dimensions.

*Parallelisation* is the most important operation of the tiling API. It corresponds to the recursive subdivision of tiles in smaller tiles, and the subsequent parallelisation of the resulting subtiles over a set of collaborating entities, such as thread blocks or warps. Consider the example in Figure 1b. A tile of size  $4M \times 4N$  is divided in subtiles, each of size

$M \times N$ . These subtiles are handled in parallel by a set of 8 cooperating warps, indicated by the numbers 0–7. The set of all cooperating warps do not need to cover the entire tile. In the example, there are 16 subtiles but only 8 warps. This means that each warp will handle 2 of these 16 subtiles. This parallelisation can be applied recursively, by dividing each of these subtiles into sub-subtiles, where each sub-subtile is handled by one thread.

*Translation* moves a tile over a specified distance in each dimension. In the example of Figure 1c, a two-dimensional tile is moved over a distance  $m$  in the  $M$  dimension, and a distance  $n$  in the  $N$  dimension. The translation operation is useful in cases where the reference point of a tile needs to be changed. For example, consider a tile referring to a submatrix stored in global memory. The coordinates of this tile are specified relative to the first element in the first row of the parent matrix in global memory. To copy this submatrix to shared memory, we need to express the tile relative to the first element stored in shared memory, which may be different. To accomplish this, we can simply translate the tile over the correct distance.

*Linearisation* is used to convert a tile's location from a Cartesian index to a linear index. This is needed to calculate the offset of a tile in memory, relative to the base pointer of the parent tile. In the example of Figure 1d, we consider a subtile at a Cartesian offset of  $(m, n)$  from its parent tile with size  $(M, N)$ . Linearisation results in the linear offset of this tile, relative to the top-left corner of the parent tile. The linearisation process assumes that the matrix is stored in column major ordering, as this is the convention that Julia uses. In this case, we need to span  $n$  columns of  $M$  elements each, and an additional  $m$  elements to reach the subtile. This corresponds to a linear index of  $nM + m$ .

### 3.3 A tiling API for Julia

To overcome challenges in developing a concrete API based on the four abstractions while meeting all requirements, we relied on some high-level Julia features.

First, a tile is fully determined by its position and its size. Our tiling API contains a `Tile` struct that stores this information. Storing the size in a field of this struct does, however, not suffice to meet the zero-cost requirement. In Julia, each function is JIT-compiled once for each combination of argument types occurring during the execution of the program. During each such JIT-compilation, no specialisation takes place based on the values of the arguments. In order for the compiler to generate high quality code for the different stages in tiled GEMMs, it needs to know how many registers are needed when transferring slices into registers. In other words, the sizes of the tiles need to be available at compile time, such that specialised code can be generated per tile size. Moreover, no dynamic type checking should be necessary in the generated code. To obtain the required specialisation, yet avoid that any dynamic type checks are needed, we defined `Tile` to be a parameterised type, where one of the type parameters (rather than a field) is the size of the tile. Julia's type inference can then obtain all the necessary information to enable specialised code generation in the JIT compiler without running into type instability issues [30].

Secondly, we observe that when we want to implement GEMM using tiling, we typically do not think in terms of

```

1 struct Tile{size, names, T}
2   base::NamedTuple{names, T}
3   offset::NamedTuple{names, T}
4 end

```

Listing 1: The definition of a Tile in the Julia tiling API.

the first or second dimension of a tile. Instead, a tile that represents a slice of the  $A$  matrix of size  $M \times K$  has  $M$  and  $K$  dimensions. Rather than writing the position as `pos[1]`, we can increase readability by naming the dimensions, so that we may write `pos.M`. This form of syntactic sugar can easily be achieved with the existing Julia type `NamedTuple`, which we use to store both the position and size of a tile.

Thirdly, we observed a form of structural bias in the Julia-LLVM tool flow with respect to address computations such as those typically occurring in recursive blocking code. The problem is that in many stages of the tiled computation, different threads operate on tiles at different positions in the input matrices or tensors. If the position stored in the `Tile` is simply a single position, unnecessarily complex PTX code is generated. However, if the position is split into a thread-dependent base index and a thread-independent offset, the compiler generates code that efficiently exploits the available register + constant addressing mode.

The final definition of the parameterised `Tile` type is shown in Listing 1. With this definition, the implementation for the translate operation is fairly simple. We define the function `translate(tile, dist)` that returns a new tile with the same size and offset, but where the base is the element-wise sum of the original tile’s base and the argument `dist`. This essentially moves the multidimensional tile over the distance specified by the argument.

The first argument of `linearise(coord, dim)` represents the coordinate of the tile. We do not take the tile itself as an argument, so that `linearise` can be used for both the base and offset of a tile. Instead of having a separate `linearise` function for base and offset, we may simply write `linearise(tile.base, ...)` and `linearise(tile.offset, ...)`. The second argument `dim` represents the size of the parent tile. To convert the Cartesian index to a linear index, we use the `LinearIndices` type from the Julia standard library. This way, we can both reuse functionality, and ensure the `linearise` operation works for any number of dimensions.

One option to project tiles is to define a function `project(tile, dims)`, where `dims` contains a list of the dimensions to keep. A projection of a tile to the  $M$  and  $N$  dimension could then be written as `project(tile, (:M, :N))`. We instead opted to use Julia’s extensibility. In Julia, the syntactic construct `a.b` is converted to a call to `Base.getProperty(a, :b)` [30]. Through the multiple dispatch mechanism, we override this function such that one can express the project operation as `tile.MN` instead of `project(tile, (:M, :N))`.

Listing 2 shows part of the implementation of the projection operation. As mentioned previously, the construct `tile.MN` is first converted to the call `Base.getProperty(tile, :MN)`. The type of the second argument, `:MN`, is a `Symbol`, indicated by the colon prefix. Symbols are similar to strings, except that they are

```

1 @inline Base.getProperty(tile::Tile{size, names, T},
2   ↪ sym::Symbol) where {size, names, T} =
3   ↪ getProperty_impl(tile, Val(sym))
4
5 @generated function getProperty_impl(
6   ↪ tile::Tile{size, names, T}, ::Val{sym})
7   ↪ where {size, names, T, sym}
8   ↪ if sym == :base || sym == :offset
9     # standard fields
10    ↪ return quote
11      ↪ getfield(tile, sym)
12    ↪ end
13  else
14    # tile projection
15    ↪ sym_str = String(sym)
16    ↪ new_names = ntuple(
17      ↪ i → Symbol(sym_str[i]), length(sym_str))
18
19    ↪ return quote
20      # create new NamedTuples with the correct dimensions
21      ↪ new_base = ...
22      ↪ new_offset = ...
23      ↪ new_size = ...
24
25      # return projected tile
26      ↪ return Tile{new_size, new_names, ...}(
27        ↪ new_base, new_offset)
28    end
29  end
30 end

```

Listing 2: Tile projection overview in our tiling API.

immutable and only one copy of each distinct value is stored [30]. The `Base.getProperty` function is specialised for arguments of type `Tile` on line 1. The value of the `sym` argument of this function determines the name of the field that was accessed. To generate custom projection implementations for each set of dimensions, we want to dispatch on the *value* `:MN` of this argument, rather than its *type* `Symbol`. To do this, we can use Julia’s `Val` type, a parametric type with one type parameter. When we call the constructor of `Val` as `Val(sym)`, a new instance of `Val` is created where the type parameter is set to `sym`. This essentially moves the value of `sym` to the type domain, so that we may use the multiple dispatch mechanism. After creating a `Val` type, we dispatch to another function `getProperty_impl` that implements the projection itself.

To make the abstraction zero-cost, we use `@generated` functions that generate custom code at type-inference time and depending on the argument types, as shown on line 3 of Listing 2. Since we moved the field name to the type domain, we can thus generate a different, specialised implementation for each projection. First note that accesses to the base or offset of a tile using `tile.base` or `tile.offset` also get converted to calls to `Base.getProperty`. Lines 4–8 handle this by checking if the passed symbol is `base` or `offset`. If so, we just return the value of the field by calling `getfield`. Julia’s `@generated` functions must return an `Expr`, which is a block of code to be compiled. Such blocks are surrounded with the `quote ... end` construct, as shown in lines 6–8.

The projection itself is implemented in lines 10–22. Line 11 converts the symbol representing the field name to a `String`, which line 12 then converts to a tuple containing the individual dimensions. For example, if `sym` is `:MN`, then `sym_str` and `new_names` are `"MN"` and `(:M, :N)`, respectively. In lines 16–18, an `Expr` is generated to create new `NamedTuples` that only contain the relevant dimensions for the base, offset, and size. Finally, line 21 wraps these newly generated `NamedTuples` in the `Tile` struct that represents the projected tile, and returns that tile.

The parallelise operation is exposed as a function call `parallelise(tile, tiling_size, index, count)`. The `tile` argument of type `Tile` is the parent tile that will be subdivided and parallelised over a set of entities that can be blocks, warps, or threads that cooperate. The second argument, `tiling_size`, determines the tile size that each entity will handle, and the last argument `count` refers to the number of cooperating entities. Finally, the argument `index` is an integer from 0 to `count - 1`, and determines the identifier of the currently executing entity.

Figure 2 shows an example parallelisation. It starts with a parent tile of size  $4m \times 2n$ , divides it in subtiles of size  $m \times n$ , and parallelises them across 2 warps. The 0/1 in each subtitle indicates the warp responsible for it. We write the operation as `parallelise(Tile(M = 4 * m, N = n), Tile(M = m, N = n), warpId, 2)`, where `warpId` is either 0 or 1, i.e. the id of the currently executing warp.

To generalise the parallelisation operation to multiple dimensions, we again reuse the indexing functionality from Julia's standard library. The information needed for iteration is then stored in a new struct, a `TileIterator`, that is returned by the `parallelise` function. Julia allows us to write customised implementations for iterating over user-defined types. For-loops are converted to calls to the `Base.iterate` function, which may be specialised for our own types. To iterate over `TileIterators` using a for loop, we must thus specialise the `Base.iterate` method for `TileIterators`. `Base.iterate` is called for each iteration of the for loop, and must return the value associated with each iteration. In the case of `TileIterators`, each call to `Base.iterate` will return a `Tile` corresponding to the tile of that iteration.

All operations on `Tiles` in our API are built on top of Julia interfaces that work for any number of dimensions. For example, the position and size of each `Tile` is stored using Julia's `NamedTuples`, which support any amount of dimensions. Similarly, the parallelisation and linearisation operations, which involve computations using multidimensional indices, are written using Julia's generic indexing interfaces. This supports higher dimensions as required.

### 3.4 Example Usage

To illustrate the use, readability and zero cost of the API, we consider three representative stages of the tiled GEMM.

#### 3.4.1 Copying a tile of $C$ from global to shared memory

To copy a tile of  $C$  from global to shared memory in step 1 of the complete GEMM, Listing 3 implements the approach illustrated in Figure 3. Each block copies a separate tile, and we launch the GEMM kernel with enough blocks to fully cover the  $C$  matrix. The tile size is determined by the `block_tile` variable. It initially has three dimensions, so we first project it to the  $M$  and  $N$  dimension using `block_tile.MN` on line 1 in Listing 3.

Next, we divide `block_tile` in subtiles and parallelise the resulting `warp_tiles` over a set of `WARPS_PER_BLOCK` cooperating warps in the block, also on line 1. The `@unroll` macro from the Julia package GPUIFYLOOPS.JL [32] informs LLVM to fully unroll the loop. Each of these `warp_tiles` has size ( $M = \text{MEM\_CD\_WARP.M}$ ,  $N = \text{MEM\_CD\_WARP.N}$ ).

Typically, `MEM_CD_WARP.N` is 1, so that the resulting `warp_tile` is highly rectangular. This is necessary to access

$m \downarrow$	$n \rightarrow$
0	0
1	1
0	0
1	1

Figure 2. Parallelisation over 2 warps each handling a  $4 \times 2$  set of subtiles.

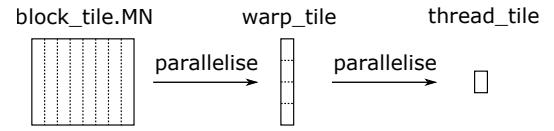


Figure 3. Copying a tile of the  $C$  matrix from global to shared memory.

global memory efficiently, as this guarantees that the threads in one warp access adjacent memory locations. The hardware is then able to coalesce these memory accesses into fewer memory transactions, thus increasing memory throughput. This is commonly referred to as *global memory coalescing*.

Similarly, we parallelise the `warp_tile` over the set of 32 threads in a warp on line 2. The integer variable `laneId` identifies the threads within a warp. Each thread handles a tile of size ( $M = \text{MEM_CD_THREAD.M}$ ,  $N = \text{MEM_CD_THREAD.N}$ ) in each iteration. In the case of an FP32 (Single Precision Floating Point)  $C$  matrix, the best choice is `MEM_CD_THREAD.M = 4`, and `MEM_CD_THREAD.N = 1`. This way, each thread loads/stores 4 adjacent FP32 elements, such that the GPU can issue one 128-bit load/store, the largest memory transaction size supported by the GPU, thus maximally vectorising the memory accesses.

The positions of all tiles are specified relative to the top-left corner of the current block's tile. This means that `thread_tile.index == (M = 0, N = 0)` corresponds to a linear index of 0. Because shared memory only stores the tile of the current block, this is the correct index for shared memory. For global memory, we need to offset this tile depending on the currently executing block. To accomplish this, we translate this `thread_tile` over the correct distance on line 3. Finally, lines 5–8 convert the base and offset of each of these `thread_tiles` to a linear index. We can then create a pointer to the correct memory location on lines 10–11, and perform the load or store. To separate the constant parts of the memory addresses, we create a pointer using the linearised base, and only add the linearised offset afterwards.

Listing 4 is equivalent to Listing 3, but does not use our tiling API. The `BLOCK_M` and `BLOCK_N` variables on line 1 are constants that correspond to the size of `block_tile` in Listing 3. The outer loop on lines 1–6 corresponds to the first parallelisation, the inner loop on lines 8–13 is the equivalent of the second parallelisation. In both loops the tile bases and offsets are calculated manually. Lines 15–18 convert the bases and offsets to linear indices, and are thus the equivalent of the linearisations in Listing 3. The translation is handled by the addition of the translation offsets `block_i` and `block_j` on line 15. Clearly the use of our tiling API in Listing 3 is less verbose and more maintainable.

Listing 5 shows the CUDA PTX to which Listing 3 is compiled. First, each thread's base addresses are computed in registers `%rd20` and `%rd13` for shared and global memory, respectively. The loads and stores are vectorised, as indicated

```

1 @unroll for warp_tile = parallelise(block_tile.MN, Tile(MEM_CD_WARP), warpId, WARPS_PER_BLOCK)
2   @unroll for thread_tile = parallelise(warp_tile, Tile(MEM_CD_THREAD), laneId, 32)
3     global_thread_tile = translate(thread_tile, (M = block_i, N = block_j))
4
5     global_linear_base = linearise(global_thread_tile.base, (M = global_M, N = global_N))
6     global_linear_offset = linearise(global_thread_tile.offset, (M = global_M, N = global_N))
7     shared_linear_base = linearise(thread_tile.base, (M = shared_M, N = shared_N))
8     shared_linear_offset = linearise(thread_tile.offset, (M = shared_M, N = shared_N))
9
10    global_ptr = pointer(global_c, global_linear_base)
11    shared_ptr = pointer(shared_c, shared_linear_base)
12
13    value = vloada(Vec{MEM_CD_THREAD, Float32}, global_ptr, global_linear_offset)
14    vstorea!(Vec{MEM_CD_THREAD, Float32}, shared_ptr, value, shared_linear_offset)
15
16 end
16 end

```

Listing 3: Copying a tile of the  $C$  matrix from global to shared memory using our tiling API.

```

1 @unroll for warp_offset = 0 : WARPS_PER_BLOCK : (BLOCK_M * BLOCK_N) ÷ (MEM_CD_WARP.M * MEM_CD_WARP.N) - 1
2   NUM_WARP_ROWS = BLOCK_M ÷ MEM_CD_WARP.M
3   base_warp_i = (warpId % NUM_WARP_ROWS) * MEM_CD_WARP.M
4   base_warp_j = (warpId ÷ NUM_WARP_ROWS) * MEM_CD_WARP.N
5   warp_i = (warp_offset % NUM_WARP_ROWS) * MEM_CD_WARP.M
6   warp_j = (warp_offset ÷ NUM_WARP_ROWS) * MEM_CD_WARP.N
7
8   @unroll for thread_offset = 0 : 32 : (MEM_CD_WARP.M * MEM_CD_WARP.N) ÷ (MEM_CD_THREAD.M * MEM_CD_THREAD.N) - 1
9     NUM_THREAD_ROWS = MEM_CD_WARP.M ÷ MEM_CD_THREAD.M
10    base_thread_i = (laneId % NUM_THREAD_ROWS) * MEM_CD_THREAD.M
11    base_thread_j = (laneId ÷ NUM_THREAD_ROWS) * MEM_CD_THREAD.N
12    thread_i = (thread_offset % NUM_THREAD_ROWS) * MEM_CD_THREAD.M
13    thread_j = (thread_offset ÷ NUM_THREAD_ROWS) * MEM_CD_THREAD.N
14
15    global_linear_base = (block_i + base_warp_j + base_thread_j) * global_M + (block_j + base_warp_i + base_thread_i)
16    global_linear_offset = (warp_j + thread_j) * global_M + (warp_i + thread_i)
17    shared_linear_base = (base_warp_j + base_thread_j) * shared_M + (base_warp_i + base_thread_i)
18    shared_linear_offset = (warp_j + thread_j) * shared_M + (warp_i + thread_i)
19
20    global_ptr = pointer(global_c, global_linear_base)
21    shared_ptr = pointer(shared_c, shared_linear_base)
22
23    value = vloada(Vec{MEM_CD_THREAD, Float32}, global_ptr, global_linear_offset)
24    vstorea!(Vec{MEM_CD_THREAD, Float32}, shared_ptr, value, shared_linear_offset)
25
25 end
26 end

```

Listing 4: Implementing the first stage in GEMM using manual calculation of addresses.

by the suffix `v4.f32`. The stores to shared memory are on lines 6, 12, and 20. As the shared memory size is known at compile time, the code exploits the register plus constant addressing modes as discussed in Section 3.3. By contrast, the compiler does not know the size of the matrix in global memory, so it does not know the linearised offset either, even though the offsets in the  $M$  and  $N$  dimensions are constants. To calculate the address in global memory, LLVM emits a multiplication (using a bit shift `shl.b64`), and an addition.

The code in Listing 5 is identical to the code that the Julia-LLVM tool flow generates for Listing 4. We conclude that no superfluous instructions are generated because of the use of our tiling API, for both the loads from global memory and the stores to shared memory.

We can use similar code for steps 2, 3-a, 3-b, 3-c-i, and 3-c-ii of the GEMM, with independently chosen tile configurations for each of them. This way, recursive double-sided blocking is supported.

### 3.4.2 Computation of the matrix product

To implement the computation of the matrix product in the inner loop using the tiling API, we will follow the approach illustrated in Figure 4. A `block_tile` represents the three-dimensional iteration space ( $M, N, K$ ) used to calculate the tile of the  $D$  matrix corresponding to one block. Let us consider the case where a `block_tile` has size  $(M, N, K) = (128, 128, 16)$ . This means that each block calculates an  $M \times N = 128 \times 128$  tile of  $D$ , by multiplying all  $M \times K = 128 \times 16$

```

1 // Calculate the base addresses in %rd13 and %rd20...
2 shl.b64           %rd22, %rd13, 5;
3 add.s64           %rd23, %rd17, %rd22;
4 cvt.a.to.global.u64 %rd24, %rd23;
5 ld.global.v4.f32  {%-f5, %-f6, %-f7, %-f8}, [%rd24];
6 st.shared.v4.f32  [%rd20+4096], {%-f5, %-f6, %-f7, %-f8};
7
8 shl.b64           %rd25, %rd13, 6;
9 add.s64           %rd26, %rd17, %rd25;
10 cvt.a.to.global.u64 %rd27, %rd26;
11 ld.global.v4.f32  {%-f9, %-f10, %-f11, %-f12}, [%rd27];
12 st.shared.v4.f32  [%rd20+8192], {%-f9, %-f10, %-f11, %-f12};
13
14 // ... repetition of similar blocks due to unrolling
15
16 mul.lo.s64        %rd64, %rd13, 480;
17 add.s64           %rd65, %rd17, %rd64;
18 cvt.a.to.global.u64 %rd66, %rd65;
19 ld.global.v4.f32  {%-f61, %-f62, %-f63, %-f64}, [%rd66];
20 st.shared.v4.f32  [%rd20+61440], {%-f61, %-f62, %-f63, %-f64};

```

Listing 5: The PTX code generated for Listing 3.

tiles in a row of  $A$  with all  $K \times N = 16 \times 128$  tiles in a column of  $B$ . These tiles of  $D$  are subsequently accumulated by summing over the  $K$  dimension.

We want to parallelise this computation over all warps in a block. In the example of Figure 5, each block contains 8 warps, in a  $2 \times 4$  arrangement. Each warp calculates a  $64 \times 32$  tile of  $D$  in each iteration, by multiplying a  $64 \times 16$  tile of  $A$ , and a  $16 \times 32$  tile of  $B$ . Of course, we want tiles in this three-dimensional space with the same  $M$  and  $N$  indices to be mapped to the same warp, so that we can accumulate across the  $K$  dimension. In the case where the matrices are stored in column-major, the warps are assigned to tiles in the order

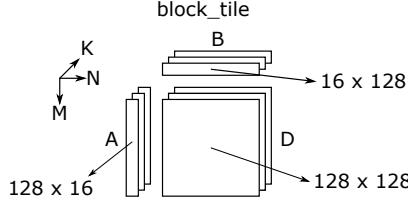


Figure 4. 3D iteration space in the inner loop of the matrix product.

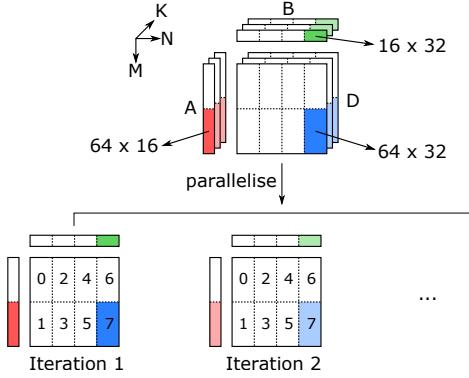


Figure 5. Computation of the matrix product in the innermost loop.

of the  $M$ ,  $N$ , and  $K$  dimension. We can thus simply use a parallelisation operation of size  $(M, N, K) = (64, 32, 16)$  across 8 warps, as shown in Figure 5. The 8 warps fully cover the  $M$  and  $N$  dimensions, as indicated by the 0–7 in each tile. In the next iteration, we have advanced along the  $K$  dimension, but the division along the  $M$  and  $N$  dimension is the same, so with this choice of tiling size, the parallelisation operation implicitly iterates over the  $K$  dimension.

Line 1 of Listing 6 shows this parallelisation operation. It returns a three-dimensional `warp_tile`. To compute the matrix product using WMMA, we first need to load the  $A$  and  $B$  tiles into WMMA fragments. To load  $A$ , we are only interested in the  $M$  and  $K$  dimension, so we first project `warp_tile` on line 3. This gives a tile of size  $(M, K) = (64, 16)$ , which thus consists of four  $16 \times 16$  WMMA fragments. To load those, we first translate the tile in the  $M$  dimension over 0, 16, 32, and 48 elements on line 3. Lines 5–6 then convert this translated base and offset to a linear index, which can then be used to create the pointer argument to `WMMA.load_a` on line 8. Lines 11–18 do the same thing for the  $B$  matrix: the `warp_tile` is projected to the  $K$  and  $N$  dimensions, translated, and converted to a linear index. Finally, lines 20–24 calculate the  $64 \times 32$  product of  $D$  using the `WMMA.mma` function from our WMMA API.

This example is perhaps the best illustration of the tiling API, as it combines all four operations on tiles: parallelisation, projection, translation, and linearisation. Using these four operations significantly improves readability compared to writing the necessary address calculations by hand.

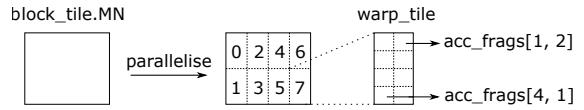
We omit the PTX code generated for this listing because it provides no additional value, but we confirm similar observations as on the first example: the base addresses of  $A$  and  $B$  for each warp are calculated once, and stored in registers. The code in Listing 6 is converted to a set of `wmma.load.a`, `wmma.load.b`, and `wmma.mma` instructions,

```

1 @unroll for warp_tile = parallelise(block_tile,
2   → Tile(M=64, N=32, K=16), warpId, 8)
3   @unroll for i = 1 : 4
4     a_tile = translate(warp_tile.MK, (M=(i-1)*16, K=0))
5     linear_base = linearise(a_tile.base, ...)
6     linear_offset = linearise(a_tile.offset, ...)
7     a frags[i] = WMMA.load_a(...)
8   end
9   @unroll for j = 1 : 2
10    b_tile = translate(warp_tile.KN, (K=0, N=(j-1)*16))
11    linear_base = linearise(b_tile.base, ...)
12    linear_offset = linearise(b_tile.offset, ...)
13    b frags[j] = WMMA.load_b(...)
14  end
15  @unroll for i = 1 : 4
16    @unroll for j = 1 : 2
17      acc frags[i, j] = WMMA.mma(a frags[i], b frags[j],
18        → acc frags[i, j], ...)
19    end
20  end
21 end

```

Listing 6: Matrix product computation using our tiling API.

Figure 6. Copying a tile of  $D$  from registers to shared memory.

and the addresses of the load operations are expressed as a constant offset from the base addresses stored in registers. This once again indicates that the tiling abstractions do not introduce any superfluous instructions.

### 3.4.3 Copying a tile of $D$ from registers to shared memory

In the previous example, we studied the calculation of the matrix product in the inner loop of GEMM. After this stage, each warp has a part of the  $D$  matrix stored in WMMA fragments. To store these WMMA fragments to shared memory, we follow the approach illustrated in Figure 6. `block_tile` represents the same tile as in example 2, i.e. the three-dimensional iteration space used to calculate a tile of the  $D$  matrix corresponding to one block. To copy  $D$ , we are only interested in the  $M$  and  $N$  dimension, so we project this tile to these dimensions first.

Next, we parallelise this tile over a set of warps. This parallelisation should have the same parameters as the matrix computation in the previous example. Obviously, the tiling size is only specified in the  $M$  and  $N$  dimension, instead of in the three dimensions  $M$ ,  $N$ , and  $K$ . Figure 6 uses the same tiling sizes as our previous example: `block_tile` is a  $128 \times 128$  matrix, and is parallelised across `WARPS_PER_BLOCK = 8` warps, each handling a `COMPUTE_WARP.M × COMPUTE_WARP.N = 64 × 32` subtile. The corresponding parallelisation operation returns a `warp_tile`, and is shown on line 1 of Listing 7. Note that the for loop of line 1 only has 1 iteration in this case, since 8 warps fully cover the `block_tile`.

Finally, this `warp_tile` is divided in a  $4 \times 2$  arrangement of WMMA fragments, like in example 2. The for loops on line 2 and line 3 iterate over these 8 WMMA fragments. Line 4 then translates the tile in the  $M$  and  $N$  dimension over 0, 16, 32, or 48 elements to obtain the final tile corresponding to each WMMA fragment. Line 6 and Line 7 then convert this Cartesian index to a linear index, so that it may be used to create pointers for the WMMA `store.d` on line 9.

```

1 @unroll for warp_tile = parallelise(block_tile.MN,
2   ← Tile(COMPUTE_WARP).MN, warpId, WARPS_PER_BLOCK)
3   @unroll for i = 1 : 4
4     @unroll for j = 1 : 2
5       tile = translate(warp_tile, (M=(i-1)*16, N=(j-1)*16))
6       linear_base = linearise(tile.base, ...)
7       linear_offset = linearise(tile.offset, ...)
8
9       WMMA.store_d(..., acc_frags[i, j], ...)
10      end
11    end
12 end

```

Listing 7: Copying a  $D$  tile from registers to shared memory.

Again, we omit the generated PTX code, but we can confirm our earlier observations. First, the base address of  $D$  for each warp is calculated and stored in a register. After this computation, 8 `wmma.store.d` instructions are emitted, which use this register as a base address, and constant offsets. Once again, we conclude that the use of the tiling API does not introduce any extra overhead.

## 4 FLEXIBLE GEMM KERNEL ABSTRACTIONS

In the previous section, we designed tiling abstractions to implement performant GEMM kernels. In this section, we add the necessary flexibility to this GEMM, such that users can instantiate a wide range of GEMM variants.

### 4.1 Requirements

The Google Brain DL (Deep Learning) research team provides an excellent overview of why this flexibility is needed [10]. They focus on Capsule networks, a novel neural network ML idea where the neurons are matrix-valued rather than scalars [33]. In short, they observed that the inflexibility of existing ML frameworks TensorFlow [34] and PyTorch [35] forced the researchers to rephrase their computations in terms of the limited set of GEMM kernels already supported by these frameworks. They had to insert multiple data transposition and matrix materialisation stages that introduced detrimental amounts of memory access overhead. They also had to insert separate kernels in between their network layers to perform simple but not-yet-established operations on the matrix elements. Not being able to fuse those operations in the GEMM kernels themselves, this again introduced massive amounts of overhead. Clearly, for advanced research in a domain such as ML, libraries providing only established GEMM functionality do not suffice.

A flexible GEMM also needs to support a multitude of different memory layouts. Basic GEMMs involve only row-major and column-major layouts. Convolutions, which are also implemented with GEMMs, involve more dimensions than matrices, so more layouts need to be considered. For images, e.g. ML frameworks typically use four dimensions: a batch of  $N$  images with  $C$  channels, each consisting of  $W \times H$  features. Among the many possible choices, NCHW and NHWC are most common [8].

Next, we consider the generalisation of matrix multiplications to multidimensional TCTs, which are common in several scientific fields, such as fluid dynamics [11], electromechanics [12], and computational chemistry [13]. Whereas a matrix-matrix multiplication has three different indices  $\{m, n, k\}$ , TCTs involve an arbitrarily large set of

indices. Matrix transpositions are extended to arbitrary permutations of those indices. The number of possible data layouts for TCTs is hence much higher. For example, a TCT of 4D tensors has a total of  $4! \times 4! \times 4! = 13824$  different memory layouts.

Given the importance of TCTs, a lot of research has been done to implement efficient support for the large number of possible cases. Springer and Bientinesi classify the traditional approaches to TCTs in three main categories [14]: loop nesting [15], [36], [37], [38], LoG [17], [18], and TTGT (Transpose-Transpose-GEMM-Transpose) [19], [20]. All three of them suffer from serious performance issues due to bad data reuse or the need to insert data reshuffling and transposition kernels.

In 2016, Springer and Bientinesi proposed another method for TCTs, GETT (GEMM-like Tensor-Tensor contraction) [38], that has since been adopted by other TCT implementations [21], [22]. GETT is based on the principles of TTGT, but implicitly reorganises tensors while loading them to avoid separate transpositions. GETT can therefore be seen as a variant of TTGT, where the transposes are fused into the GEMM kernel. Clearly, this fusion requires that the underlying GEMM kernel is flexible.

While the tiling and WMMA APIs introduced in previous sections allow that flexibility, programming against them would still be quite cumbersome. We hence propose a higher-level API based on a high-performance GEMM kernel that can easily be customised through a set of higher-level abstractions that are as intuitive as possible to researchers from, e.g. the domains of ML and DL. We put forward the following main requirements for this high-level GEMM API:

- *Flexibility* with respect to data layouts, on-the-fly data transpositions, and fused operations are prime objectives. Support for other, more complex data types such as complex numbers [2] or dual numbers as used in automatic differentiation [39] is also of interest.
- *Performance* of GEMM kernels that are built using our API should be on-par with the state-of-the-art implementations, such as CUBLAS or CUTLASS. This obviously implies that our GEMM should be WMMA-compatible and support double-sided recursive blocking, i.e. independent tiling parameters need to be supported for the data transfer stages at different levels of the memory hierarchy and for the computational stages.
- *Portability*: GEMM kernels built with our API should perform well on a range of devices. We should hence make as few assumptions about the underlying hardware as possible. For example, our API needs to be able to handle different shared memory sizes, as well as GPUs with and without TCs of different generations.

### 4.2 A flexible, abstract GEMM API for Julia

Our strategy is to implement the general structure of a performant GEMM kernel beforehand. To make it flexible, we split this GEMM in a small set of building blocks with a predetermined interface. Concretely, the GEMM contains calls to a set of functions with a predetermined name. To extend the basic implementation, it will suffice to implement new versions of the called functions that customise the behaviour based on their input types. Julia's

just-in-time type inference and compilation flow enables us to perform this split without introducing performance overhead. Furthermore, Julia's multiple dispatch allows us to make the split orthogonally, which results in more intuitive building blocks and eases code reuse and hence programmer productivity.

#### 4.2.1 Params

We of course still want the user to be able to customise the tiling size of each step of the GEMM kernel. This is the purpose of the `params` abstraction. This abstraction is essentially a structure that is passed to the kernel, and contains a set of configuration fields. Some of these fields determine the tiling sizes, others specify the kernel's launch configuration, such as the number of warps per block. The user does not need to specify all fields manually. We have implemented a set of heuristics that choose reasonable defaults for fields that are not set explicitly. For example, if the tiling size per threadblock is not set, we choose the largest square ( $N \times N$ ) or nearly-square ( $2N \times N$ ) tile that still fits in shared memory. For the time being, these heuristics are mainly aimed at GEMMs using TCs, but future work could expand these heuristics to other cases as well.

#### 4.2.2 Layouts

The positions of tiles at different levels in the memory hierarchy in our tiling API are expressed in logical coordinates. To convert these logical coordinates to offsets in physical memory, we introduce another abstraction, called *layouts*. This abstraction corresponds to three functions that can be customised using Julia's multiple dispatch. The `size(layout_type, logical_size)` function determines the size in physical memory of the layout for a given size in logical coordinates. This physical size is not necessarily the same as the size in logical coordinates. For example, to access shared memory efficiently, it is sometimes necessary to add  $p$  padding elements to every column of a column major matrix. In this case, for a logical size of  $M \times K$ , the corresponding physical size would be  $(M+p) \times K$ . The `size(...)` function is used so that our GEMM API knows how many bytes it has to reserve in shared memory. This function is also used by the heuristics in the `params` abstraction to select the optimal tiling size in shared memory, as this depends on how much memory a given memory layout requires.

The other two functions are `load(layout_type, tile, ...)` and `store(layout_type, tile, ...)`. As their name suggests, these functions are responsible to load or store the tile at the logical coordinates represented by the `tile` argument. With these functions, users can implement arbitrary logic to load or store the matrix elements corresponding to a given tile. For example, recall that NVIDIA GPUs can load and store vectors of 16 bytes (128 bits) in a single instruction. This vectorisation of memory accesses is only possible if the base address of the load or store is aligned, i.e. divisible by 16. An `AlignedColumnMajor` layout can indicate that the necessary alignment requirements are met, so that the corresponding `load` and `store` functions can issue vectorised loads and stores.

For a classic GEMM kernel, the most obvious instantiations of the layout building block are `RowMajor` and `ColumnMajor`. As mentioned before, each of these can be

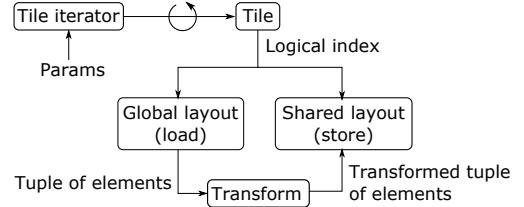


Figure 7. Copying an  $A$  tile from global to shared memory.

adapted to aligned or padded layouts. To add padding, one could have `PaddedRowMajor` and `PaddedColumnMajor` layouts, but Julia's type system allows us to do this more cleanly. We can make a parameterised type `PaddedLayout {Layout, Padding}`, where `Padding` represents the padding in number of elements, and `Layout` is the base layout we wish to modify, such as `RowMajor` or `ColumnMajor`. The `load` and `store` functions for padded layouts would then dispatch to the implementations for the underlying `Layout`.

The layout building block can also be used to create a GEMM with a more complicated mapping between logical indices and physical offsets. For example, GETT's reinterpretation of multidimensional tensors as matrices can be performed using a custom implementation of the layout building block. Note that a layout does not even need to correspond to a matrix that is materialised in memory. Consider a matrix multiplication where the elements of one of the matrices can be calculated from the position, i.e.  $a_{ij} = f(i, j)$  for some function  $f$ . In this case, we implement a layout where the `store` function is a no-op, and the `load` function generates the necessary elements on the fly. Similar strategies can be used for other matrices with a special structure, such as sparse matrices or diagonal matrices. We can only store the non-zero elements in memory, and create a custom layout that implements the necessary logic to load or store the correct elements.

#### 4.2.3 Transforms

The next building block is that of *transforms*. Transforms are arbitrary Julia functors, i.e. functions or structures implementing the function call operator `()`. They are called after every load and before every store operation in the GEMM. By having a transform after every load and before every store, element-wise operations to the input and result matrices can be applied consistently in our API.

Transforms can serve for element-wise operations, such as a simple scaling in the case of GEMM, and for activation functions for artificial neurons in neural networks. Another use case of transforms is to implement type conversions immediately after loading data from global memory. This is useful if one wants to use a higher precision data type to compute the GEMM, but store the matrices in lower precision in global memory to save capacity.

Figure 7 illustrates how `params`, `layouts`, and `transforms` interact to copy a tile of the  $A$  matrix from global to shared memory. A similar structure is used to copy tiles of the  $B$ ,  $C$ , or resultant  $D$  matrix. This copy operation is performed cooperatively by all threads in a threadblock, using the parallelisation operation of our tiling API. First, the `params` component determines the tiling size that should be used

for the tile iterator corresponding to the parallelise operation. The GEMM kernel then iterates over this tile iterator, which returns a tile in each iteration. The base and offset of this tile are specified in logical coordinates. To load the correct matrix elements from global memory, the `load` function is called using this tile and the layout of  $A$  in global memory. This `load` function returns a tuple that contains the correct matrix elements. This tuple is then sent to the transform for the global-to-shared stream of the  $A$  matrix, resulting in a transformed tuple. Finally, the `store` function corresponding to the layout of  $A$  in shared memory is called with this transformed tuple and the logical index of the current tile.

#### 4.2.4 Operators

The previous building blocks together copy tiles from global to shared memory. The purpose of the next building block, called *operators*, is to load tiles from shared memory, perform the matrix multiplication, and store the resulting tile back to shared memory. To do so, this building block has five functions associated with it.

The `load_a`, `load_b`, and `load_c` functions load tiles of the  $A$ ,  $B$ , and  $C$  matrix from shared memory to registers. The matrix computation itself is performed by the `mma` function, and the result is stored back to shared memory using the `store_d` function. Like the layout building block, the `load_a`, `load_b`, `load_c`, and `store_d` functions have a `tile` argument that represents the logical coordinate of the tile that should be loaded or stored. The load and store functions also have an argument that determines the shared memory layout of the corresponding matrix, so that we can dispatch to the different implementations depending on the memory layout that is used. Finally, the `mma` function has three arguments `a_frag`, `b_frag`, and `c_frag` that represent parts of the  $A$ ,  $B$ , and  $C$  matrices stored in registers. The function should perform the multiply-accumulate operation `res_frag = a_frag * b_frag + c_frag`, and return the resulting fragment `res_frag`.

The listed functions map one-to-one onto the steps of the WMMA API mentioned in Section 2.3. This is no coincidence, as both the operator building block and the WMMA API are warp-level matrix multiply-accumulate operations. It is hence fairly easy to define an implementation of the operator building blocks that uses TCs using our WMMA API. It suffices to convert the `tile` argument to the load and store functions to a memory address, and call the `load`, `store`, and `mma` functions of the WMMA API.

The operator building block has several use cases. First, it can be used to provide a custom implementation for the computation in the inner loop of GEMM. This is useful if the data type of our matrices has a custom multiplication operator, such as complex numbers or dual numbers. The operator building block also improves the portability of the GEMM kernel. For example, the WMMA operator may be parameterised with the WMMA shape, so that we can select the WMMA shape that is optimal for our GPU. Alternatively, we can define an alternative operator that calculates the matrix product using the traditional FPUs (Floating Point Units) instead of TCs on devices that lack the latter.

---

```

1 @unroll for warp_tile = parallelise(block_tile.MN,
2   ↪ Tile(MEM_CD_WARP), warpId, WARPS_PER_BLOCK)
3   @unroll for thread_tile = parallelise(warp_tile,
4     ↪ Tile(MEM_CD_THREAD), laneId, 32)
5     global_thread_tile = translate(thread_tile,
6       ↪ (M=block_i, N=block_j))
7     x = Layout.load(GLOBAL_C_LAYOUT, c, global_thread_tile)
8     y = transform_global_to_shared_c(x, thread_tile)
9     Layout.store(SHARED_C_LAYOUT, shmem_c, y, thread_tile)
10    end
11  end

```

---

Listing 8: Copying a  $C$  tile from global to shared memory.

#### 4.2.5 Epilogues

While the already discussed transform abstraction already allows performing element-wise operations on the  $D$  matrices, we add an epilogue abstraction to our API to enable customisation of the way global memory is updated at the last stage of GEMM. This enables, e.g. to apply a reduction operation across all threadblocks.

In the general GEMM implementation, our epilogue building block only has one purpose: to copy tiles of the resultant matrix from shared memory to global memory. By default, we only include one epilogue that simply copies the current threadblock's tile in shared memory to the correct position in global memory. This default epilogue uses the previously mentioned layout building block to determine the memory layout of the resultant  $D$  matrix.

### 4.3 Example Uses

#### 4.3.1 Fully-featured interface

As a first example of how to use the presented building blocks and API, we consider the first step in a GEMM kernel: copying a tile of the  $C$  matrix from global to shared memory. Listing 3 showed an implementation of this step using our tiling API. In our GEMM API, this first step is implemented as shown in Listing 8. The code has a similar structure to Listing 3, but the linearisation, loads, and stores are replaced by generic calls to `Layout.load` and `Layout.store`. The first arguments of these functions, `GLOBAL_LAYOUT` and `SHARED_LAYOUT`, are types that determine the memory layout of  $C$  for global and shared memory, respectively. The `transform_global_to_shared_c` is a Julia function that represents the transform that should be applied during the global-to-shared memory stream of the  $C$  matrix.

Now suppose that we have defined the necessary components (such as layouts, operators, ...) for a given use case. To instantiate and execute GEMM kernels that use these components, we use the user-facing interface of our GEMM API, which is illustrated in Listing 9. This code fragment calculates a mixed-precision matrix product of the form  $D_{ij} = \max(\sum_k A_{ik}B_{kj} + C_{ij}, 0)$ . These types of matrix products are common in neural networks, where the activation function  $\max(\cdot, 0)$  is commonly referred to as a rectified linear unit (ReLU). Lines 1–4 declare the two-dimensional arrays that represent the  $A$ ,  $B$ ,  $C$ , and  $D$  matrices. In lines 6–11, we configure the parameters of our GEMM kernel, such as the overall shape of the GEMM, the operator to be used in the inner loop, and the memory layouts of the  $A$  and  $C$  matrices. The missing fields are automatically set to reasonable default values. For example,

```

1 a = CuArray(rand(Float16, (M, K)))
2 b = CuArray(rand(Float16, (K, N)))
3 c = CuArray(rand(Float32, (M, N)))
4 d = similar(c)
5
6 conf = GemmKernels.get_config()
7 gemm_shape = (M = M, N = N, K = K),
8 operator = Operator.WMMAOp{16, 16, 16},
9 global_a_layout = Layout.AlignedColMajor{Float16},
10 global_c_layout = Layout.AlignedColMajor{Float32}
11
12
13 GemmKernels.matmul(
14   a, b, c, d, conf;
15   transform_regs_to_shared_d = Transform.Elementwise(
16     → x -> max(x, 0)
17   )

```

Listing 9: Matrix product  $D_{ij} = \max(\sum_k A_{ik} \cdot B_{kj} + C_{ij}, 0)$ .

```

1 a = CuArray(rand(Float16, (M, K)))
2 b = CuArray(rand(Float16, (K, N)))
3 c = CuArray(rand(Float32, (M, N)))
4
5 alpha = rand(Float32)
6 beta = rand(Float32)
7
8 GemmKernels.BLAS.gemmEx!('N', 'N', alpha, a, b, beta, c)

```

Listing 10: A matrix product using our BLAS-like interface.

if the memory layout of the  $B$  matrix is not specified, it is automatically set to the memory layout of the  $A$  matrix.

A GEMM kernel using this configuration is executed in lines 13–16. The transform that should be applied when copying tiles of the resultant  $D$  matrix from the register file to shared memory is determined by the argument `transform_regs_to_shared_d`. The call to `GemmKernels.matmul` will execute each step in the GEMM kernel, using the components given by the user. For example, Listing 8 will be executed with `GLOBAL_C_LAYOUT = Layout.AlignedColMajor{Float32}`. We conclude that we can instantiate and launch customised GEMM kernels easily, without sacrificing flexibility.

#### 4.3.2 BLAS-like interface

The interface of the previous section exposes maximal flexibility to the user, but differs from the interface used by CUBLAS. We also provide a more familiar BLAS-like interface which can be used if not all flexibility is needed. This interface supports all operations of CUBLAS’s `gemmEx`, i.e. linear scaling and transposition of the input matrices, but, importantly, with support for many more input types.

To ease the transitioning process, this BLAS-like interface has the same signature as CUDA.JL’s `gemmEx` wrapper. To use our GEMM kernels in existing code, it suffices to replace `CUDA.CUBLAS.gemmEx!` by `GemmKernels.gemmEx!`.

Using the BLAS interface, there is no need to specify each component manually. Instead, they are derived from the types of the arguments. For example, Listing 10 calculates the matrix product  $C := \alpha \cdot A \cdot B + \beta \cdot C$ . Based on the combination of the types of the  $A$ ,  $B$ , and  $C$  matrix, our implementation of the BLAS-like interface instantiates a GEMM kernel with `operator = Operator.WMMAOp{16, 16, 16}`, `global_a_layout = Layout.AlignedColMajor{Float16}`, etc.

Using the BLAS interface in library code allows extending this library with new types, without changing the library code. For example, the library code in Listing 11 is called

```

1 function library_code(a::CuArray, b::CuArray, c::CuArray)
2   # ...
3   GemmKernels.BLAS.gemmEx!('N', 'N', alpha, a, b, beta, c)
4   # ...
5 end

```

Listing 11: Library code making use of our BLAS-like API.

```

1 for f in (:load_a, :load_b, :load_c, :store_d)
2   @eval @inline $f(op, ::Type{Layout.Padded{L, P}}), args...)
3   ↪ where {L, P} = $f(op, L, args...)
3 end

```

Listing 12: Redirecting operator calls for padded layouts to the underlying layout, using Julia’s metaprogramming.

for GPU arrays, but does not impose any restrictions on the element type. Using custom element types is as simple as adding support for them in our GEMM framework, and calling the library code with this new type.

## 4.4 Discussion

For our GEMM framework design, we have strived for orthogonality of different components. For example, epilogues and operators only interact via shared memory and can be combined arbitrarily as long as they both support the same shared memory layout. Transforms use broadcast expressions that work for different data types and array lengths, and can hence be combined with different layouts or params.

Nevertheless, some inevitable coupling between different components remains. Most prominently, layouts are coupled to epilogues (e.g. a bias epilogue can require different logic for row-major and column-major layouts), and to operators (e.g. WMMA supporting padded and non-padded layouts). Luckily, we can reduce the impact on code verbosity and reuse through several features of Julia. Epilogues can contain layout-agnostic code, and rely on fine-grained method overloading for layout-specific code paths. Metaprogramming can be used to redirect operator calls for padded layouts to the underlying layout, as illustrated in Listing 12.

Another point that merits some discussion is the extent to which our APIs and abstractions are Julia-specific, i.e. whether or not parts of them can be used for similar APIs in other PLs. While none of our proposed abstractions are Julia-specific, implementations in other PLs would suffer from reduced code reuse, or increased overhead and verbosity. Julia’s unique combination of multiple dispatch, type inference, and JIT compilation allows us to compose GEMM operations from different components, without incurring any run time overhead. Due to the parametric nature of Julia’s type system, tile sizes can be moved to the type domain, such that specialised code can be generated per tile size. Julia’s metaprogramming capabilities prove extremely powerful to improve code reuse and reduce code verbosity.

It is also possible to call Julia functions from C code, using Julia’s C API. Since most PLs can call C functions, our framework’s kernels can be used in C++, Python, C#, and other high-level PLs, and hence also in ML frameworks written in these PLs, such as TensorFlow and PyTorch.

Finally, we should discuss the issue of portability. So far, we focused on flexible GEMMs for CUDA-enabled GPUs. Nevertheless, the abstractions in our tiling API and

flexible GEMM API are vendor-agnostic, and we expect they can be reused for AMD and Intel GPUs. More concretely, porting our framework necessitates two changes. First, our WMMA operator needs to be replaced with an operator using traditional floating point hardware. Secondly, our template kernel contains CUDA-specific concepts such as `threadIdx`, and hence needs to be ported to OPENCL. Code duplication can be avoided using the package `KERNELABSTRACTIONS.JL` that allows writing vendor-agnostic GPU kernels [40].

## 5 EVALUATION

To evaluate the performance and flexibility of our APIs, we created the necessary components for five GEMM variants as discussed below. Run times were measured on an NVIDIA RTX 2080 Ti with NVIDIA Nsight Compute and with `BENCHMARKTOOLS.JL`, which continues sampling until the standard deviation becomes small enough. We compare the performance of our kernels to CUTLASS 2.2, CUTENSOR 1.3.0 and CUBLAS 11.2. We set the latter's math mode to `CUBLAS_TENSOR_OP_MATH` and call `cublasGemmEx`. We use CUDA 11.0, CUDA.JL 2.0, and Julia 1.5.

### 5.1 Mixed-precision GEMM

Our first example is a normal mixed-precision GEMM, i.e. a computation of the form  $D = A \cdot B + C$ . This operation is directly supported by NVIDIA's CUBLAS library. To use TCs in our GEMM framework, we create an operator that simply calls the correct WMMA functions in our WMMA API for each step in the GEMM's inner loop.

In a GEMM, the  $A$  and  $B$  matrices may be stored in a column-major memory layout (N), or a row-major memory layout (T). We hence implemented both a `ColMajor` and `RowMajor` layout component. These layouts are suitable for global memory, but lead to inefficient memory accesses in shared memory. On NVIDIA GPUs, shared memory is split into a set of *memory banks*. Memory accesses to addresses that map to the same bank, so-called bank conflicts, are serialised.

The simplest way to reduce these bank conflicts is to add padding to every column or row, such that the mapping of matrix elements to banks is changed. To achieve this, we use a `PaddedLayout` component to store matrices in shared memory. This layout serves as a wrapper for other layouts, e.g. `PaddedLayout{ColMajor, 8}` is a column-major layout, where every column is padded by 8 elements.

Out of the three functions associated with layouts, only `size` needs to be specialised for each type of padded layout. This is necessary because padding differs for, e.g. row-major layouts vs. column-major layouts. Calls to `load` or `store` are automatically redirected to the underlying layout. As a result, supporting padding only required adding 14 lines of source code. To use padded layouts for a GEMM, it suffices to set, e.g. `shared_a_layout = Layout.PaddedLayout{Layout.ColMajor, 8}`, similarly to Lines 9–10 in Listing 9.

The epilogue for mixed-precision GEMM simply copies a tile from shared memory to global memory.

Figure 8 compares the performance of our mixed-precision GEMM to CUTLASS and CUBLAS. The different lines and markers represent the four combinations of the

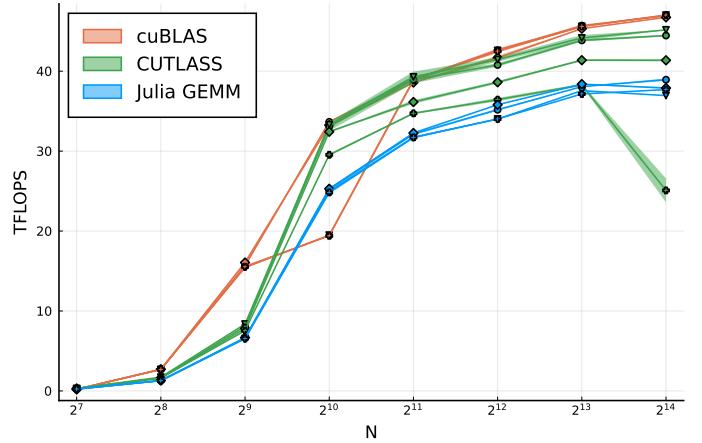


Figure 8. Performance of mixed-precision GEMM.

data layouts of the  $A$  and  $B$  matrices, the shaded regions represent error margins. We have no explanations for the two anomalous results in the measurements for CUTLASS and CUBLAS. They occurred consistently over many experiments. For the most interesting, larger matrices with  $N=2048$  and more, the performance of our kernels ranges between 82% and 86% of CUBLAS, the best performing library. We conclude that our kernels achieve reasonably good performance, despite being written completely in Julia. To the best of our knowledge, no existing implementation purely written in a single higher-level PL comes close.

The performance difference between our kernel and the state-of-the-art in CUTLASS and CUBLAS can be attributed to two factors. First, our implementation does not yet contain data swizzling to avoid bank conflicts in shared memory. A device-dependent layout is necessary for swizzling optimally for a GPU's shared memory implementation, which we have not yet explored. Secondly, CUBLAS does not use WMMA, but accesses TCs directly, allowing the use of custom memory layouts in shared memory that perform better than our padded layouts. Implementing this in our framework necessitates a new layout and a new operator component. This custom operator would use the `mma` family of instructions instead of WMMA. Contrary to WMMA, `mma` does not have load or store instructions, so distribution of matrix elements to different threads needs to be done explicitly. Use of the `mma` operator would hamper portability, though, because some `mma` instructions are optimised for a particular GPU architecture. Engineering this is future work.

### 5.2 Diagonal matrices

Next, we focus on a mixed-precision GEMM where the  $A$  or  $B$  matrix is a diagonal matrix. In Julia, diagonal matrices are represented using the `Diagonal` parametric type. This type acts as a wrapper for a one-dimensional array, which contains the diagonal elements. References to elements on the diagonal are redirected to loads or stores of this underlying array, whereas references off the diagonal return 0 without performing any actual memory access.

We leveraged Julia's multiple dispatch to provide a GEMM implementation that is specialised for diagonal matrices by means of two optimisations. First, we replace the layouts from Section 5.1 with a `Diagonal` layout. Similar

Table 1

Performance of our diagonal matrix GEMM and the equivalent cuBLAS implementation, for  $N = 4096$ ; run times are for 100 iterations.

	cuBLAS	Ours
Run time (ms)	$322.77 \pm 0.08$	$50.68 \pm 0.05$
#Global mem. accesses	3410K per kernel	490K per kernel
#Tensor Core instr.	67 109K per kernel	3113K per kernel

to Julia’s Diagonal wrapper, this layout simply returns 0 if the accessed element lies off the diagonal, and otherwise accesses an array. Second, we extended our template GEMM kernel with a customisable predicate that determines for each inner loop iteration if it should be executed or skipped. By default, this predicate is constant `true`, but we specialise it for diagonal matrices so that iterations that perform computations on elements off the diagonal are skipped.

Table 1 compares the performance of this specialised GEMM kernel with cuBLAS. As cuBLAS does not include GEMM kernels specialised for diagonal matrices, the cuBLAS version of our code had to materialise the diagonal matrix before calling the standard cuBLAS GEMM kernel. This is in line with the common practice as discussed in Section 4.1. The first optimisation results in a reduction of 89% in global memory traffic. The second optimisation reduces the number of TC operations by over 95%. Together, they lead to a GEMM that is more than 6 times faster than what we can obtain with cuBLAS’s inflexible kernels.

Adding support for diagonal matrices required adding 23 lines of source code to our existing framework. We conclude that specialisation of kernels in our framework requires minimal effort, while at least in some cases still obtaining massive performance improvements.

### 5.3 Operator fusion

cuBLAS fuses linear scaling into its GEMM computation, i.e. its GEMM is of the form  $D = \alpha \cdot AB + \beta \cdot C$ . Other computations cannot be fused in cuBLAS’s kernels and require a separate kernel launch. We consider two examples of GEMM computations that can exploit operation fusion: custom element-wise operations and adding a bias vector.

In custom element-wise operations, the linear scaling with  $\alpha$  and  $\beta$  is replaced by any arbitrary function. We implemented this in our GEMM framework using an `ElementwiseTransform` component. It has an arbitrary function as a parameter, which is applied to every element.

In bias computations, a one-dimensional bias vector is added to every row of the matrix product. To add support for bias in our GEMM framework, we created a custom epilogue that loads the bias vector from global memory, and adds it to the matrix product in shared memory, before writing the result back to global memory.

Table 2 compares the run times of six GEMMs with and without element-wise operations on input and/or output matrices, and with and without bias vectors. For the purpose of this experiment, we use additions with a constant and ReLU, a popular function in the domain of ML, as the element-wise operations, but similar results are obtained with other ones that are not supported in CUTLASS, and for which researchers would have to fall back on cuBLAS or our solution. For this reason, we only compare to cuBLAS.

Table 2

Run times of cuBLAS that lacks fusion capabilities and our GEMMs that exploit operation fusion, for  $N = 4096$  and 100 iterations.

	cuBLAS [ms]	Ours [ms]
GEMM	$260.82 \pm 1.51$	$312.22 \pm 1.61$
GEMM + ReLU on D	$286.20 \pm 1.10$	$313.00 \pm 1.53$
GEMM + bias	$287.72 \pm 1.80$	$315.56 \pm 1.57$
GEMM + bias + ReLU on D	$288.19 \pm 2.15$	$315.54 \pm 1.60$
GEMM + bias + ReLU on C & D	$313.88 \pm 1.22$	$315.40 \pm 1.95$
GEMM + bias + ReLU on C & D + addition operation on A & B	$340.38 \pm 1.21$	$316.73 \pm 1.84$

With cuBLAS, combining the GEMM with element-wise operations or bias vectors results in additional kernel launches. While we have fused the operations as much as possible, i.e. the application of ReLU on  $D$  and the addition of the bias vector are fused in one kernel, the GEMM and the other operations each still correspond to separate kernels which cannot be fused any further. By contrast, our framework seamlessly fuses all operations in the GEMM kernel instead. The effect is clearly visible in the results in the table. While the standard GEMM in cuBLAS is faster than with our framework, in line with the results in Section 5.1, our framework catches up as more fusible operations are added. Whereas the extra operations require almost no additional time in our framework (bias vectors need to be loaded, hence their small cost), each operation requiring a separate kernel costs approximately 10% in performance with cuBLAS. Ultimately, our GEMM becomes about 7% faster. This clearly illustrates the need for operator fusion, and how effective our approach is. The fact that our approach fuses the operations seamlessly also implies that any future optimisation of our default implementation of the mixed-precision GEMM, through swizzling or use of mma instead of WMMA, will automatically benefit GEMMs with fusible operations as well. Our GEMM will then outperform cuBLAS even more.

Note that, although our Julia GEMM implementation can easily be invoked from within other languages and ML frameworks, as discussed in Section 4.4, this advantage of fused element-wise operations can only be obtained if they are expressed in Julia, because that fusion depends on the Julia-specific features also discussed in Section 4.4.

### 5.4 Complex and dual numbers

Standard mixed-precision GEMMs differ from mixed-precision GEMMs of complex numbers in two ways. First, the WMMA multiply-accumulate operation in the inner loop is replaced by four WMMA operations: `A.real * B.real`, `A.real * B.imag`, `A.imag * B.real`, and `A.imag * B.imag`. Second, complex GEMMs use different memory layouts in global and shared memory. In global memory, complex matrices are typically stored in an interleaved layout, where the real and imaginary parts are stored contiguously. This layout is incompatible with WMMA, so in shared memory, we use a split layout instead, where the real and imaginary parts are stored separately. In our GEMM framework, these two differences correspond to a new operator `WMMAComplexOp`, and two new layouts `InterleavedComplex` and `SplitComplex`, respectively.

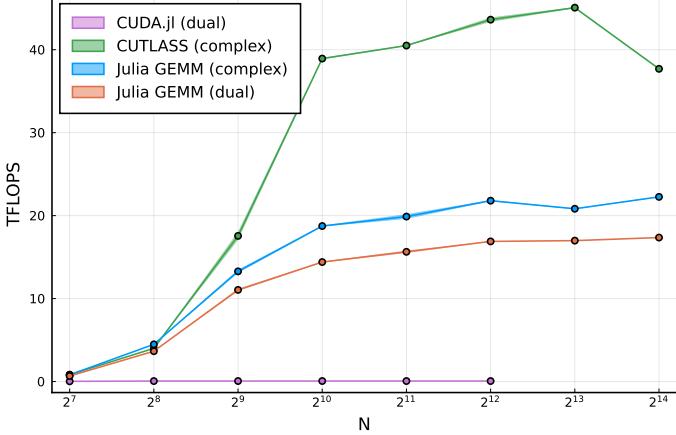


Figure 9. Performance of complex and dual GEMMs.

Dual numbers differ slightly from complex numbers. The imaginary unit  $i$  is replaced by  $\varepsilon$ , and  $\varepsilon^2 = 0$  whereas  $i^2 = -1$ . As such, we need an additional `WMMADualOp` operator component, but we can reuse the split and interleaved layouts we developed for complex matrices.

Figure 9 shows the performance of four GEMM kernels using complex or dual numbers. As CUBLAS supports neither complex numbers using TCs nor dual numbers, we did not include it in the comparison. CUBLASLT does support complex numbers, but uses CUTLASS's kernels. CUTLASS only supports complex numbers. So we compare the performance of our two kernels to CUTLASS for complex numbers and to the generic CUDA.jl kernel that is invoked when our API is not used to compute a mixed-precision GEMM on dual numbers. Our complex number implementation achieves a peak performance of 59% of CUTLASS's peak performance. We conclude that, despite the fact that our kernels are written completely in Julia, and do not contain optimisations specific to complex GEMM, we are still able to reach reasonably good performance.

The difference in performance between our complex GEMM and CUTLASS's can partly be attributed to CUTLASS's use of a split layout in both global and shared memory, eliminating the overhead of changing layouts in the global-to-shared memory stream. Our kernels use an interleaved layout in global memory, as this is the format that Julia uses. This comes with some overhead, but eliminates the need for extra interleaved-to-split and split-to-interleaved kernel launches that would be necessary for CUTLASS.

While CUDA.jl contains wrappers for CUBLAS's GEMM kernels, it supports only a limited number of data types. Calling these wrappers with unsupported types, such as dual numbers, falls back to a generic implementation that is many orders of magnitude slower, as is clear from the CUDA.jl line in Figure 9.

Adding support for complex and dual numbers to our framework required 169 lines of source code, of which 85 are common to both data types. We conclude that extending our framework with custom data types requires minimal effort, especially compared to the alternative of writing a performant GEMM kernel for these data types from scratch.

```

1 abstract type LayoutA{T} <: Layout.AlignedColMajor{T} end
2
3 @inline function Layout.load(::Type{LayoutA{T}}),
4   workspace, tile::Tile{size}) where {T, size}
5   NUMEL = 16 ÷ sizeof(T)
6
7   M = tile.base.M + tile.offset.M
8   K = tile.base.K + tile.offset.K
9   d = K
10  a = M ÷ Base.size(workspace, 1)
11  b = M % Base.size(workspace, 1)
12  offset = 1 + b + d * Base.size(workspace, 1) +
13    a * Base.size(workspace, 1) * Base.size(workspace, 2)
14  Layout.vloada(Layout.Vec{NUMEL, T}, pointer(workspace),
15    offset)
16 end

```

Listing 13: Layout for tensor A.

Table 3  
Performance of TCT and the equivalent CUTENSOR implementations, for  $N_a = 64, N_b = 32, N_c = 2048, N_d = 2048$ ; run times are in  $\mu\text{s}$ .

cuTENSOR TTGT	cuTENSOR GETT	Ours
$351.14 \pm 2.46$	$1238.81 \pm 9.39$	$433.82 \pm 5.26$

## 5.5 Tensor Contraction

Implementing and evaluating fully-optimised general TCTs is out-of-scope of this paper. However, we do want to demonstrate that our GEMM functionality does provide the necessary support for common TCTs. We hence evaluated the functionality for one TCT, namely the first benchmark of the Tensor Contraction Code Generator benchmarks [16]. In Einstein notation, this TCT is written as  $D_{abc} = A_{bda} \cdot B_{dc}$ .

Using our GEMM APIs, we created custom global memory layouts for matrices  $A$  to  $D$ . The layout for  $C$  always returns zero without performing loads to global memory. The layouts for  $A$ ,  $B$ , and  $D$  provide the fused transpositions of GETT [38]. As the contraction index  $d$ 's position is almost perfect for an NN-GEMM, the  $A$ ,  $B$ , and  $D$  layouts started from the `AlignedColMajor{T}` layouts. Custom mappings from elements in global memory to shared memory were implemented such that GEMM in shared memory is equivalent with TCT in global memory. We did not extensively search for optimal memory layouts, but simply tried to use as many stride-1 accesses in global memory as possible, to facilitate vectorisation and global memory coalescing. As an example, the  $A$  layout code is shown in Listing 13. With this layout,  $A_{bda}$  in global memory becomes  $A_{bad}$  in shared memory, which is then interpreted as the 2D matrix  $A_{mk}$ , by mapping of dimensions  $(ba) \rightarrow m$  and  $d \rightarrow k$ .  $B$  is  $B_{dc}$  in global memory and stays the same in shared memory, with mapping  $d \rightarrow k$  and  $c \rightarrow n$ . The GEMM in shared memory then computes  $A_{mk}B_{kn} = A_{bad}B_{dc} = D_{bac}$ , and the custom  $D$  layout ensures shuffling is performed to  $D_{abc}$  while copying data from shared to global memory.

We compare the performance of our implementation to that in CUTENSOR, a hand-optimised tensor library from NVIDIA. We observed that CUTENSOR by default does not use the GETT approach for the evaluated TCT, as it executes two kernels, similar to the TTGT approach [19], [20]. We can however force it to use the GETT approach as well.

Table 3 lists the performance results. Our kernels achieve 81% of the performance of CUTENSOR TTGT, the best performing configuration. Interestingly, CUTENSOR's GETT

kernel performs  $3.5\times$  worse than TTGT, and we outperform it by a factor of 2.9.

The  $B$  and  $D$  layouts look similar to Listing 13; the  $C$  layout always returns 0 and elides the indexing logic and memory operations, but otherwise has the same structure. In total, these four layouts add up to 58 lines of code.

We conclude that our framework can support (at least some variants of) TCTs, and achieves performance in the same ballpark as that of state-of-the-art kernels in CUTENSOR, while requiring minimal programming effort.

## 6 RELATED WORK

In Section 4.1, we already discussed the need for flexible and performant GEMMs on GPUs in various domains. As discussed in Section 2.3, several libraries exist to deliver the performance. We compared the performance of our GEMM API with some of them in Section 5.

NVIDIA’s CUTLASS template library contains components to instantiate performant GEMMs on GPUs [41]. As it is written in C++, it does not solve the two-language problem and its impact on programmer productivity. It did serve, however, as an inspiration for the abstractions in our GEMM API. For example, CUTLASS also has the notion of layouts that map logical indices to physical memory offsets, and epilogues for custom post-processing. Some components have a slightly different purpose, however. In CUTLASS, transforms only apply to the global-to-shared memory stream of the  $A$  and  $B$  matrices. Element-wise transforms on the resultant matrix are handled in a custom epilogue. Adding support for custom transformations requires significant effort, as CUTLASS epilogues are typically 150–200 lines long. In our GEMM API, transforms are applied after every load and before every store, ensuring that element-wise operations to the input and resultant matrices can be applied more easily and consistently.

The CUTLASS codebase is extensive and contains many components, making it more difficult for end users to get started extending it. Each GEMM typically involves quite a few layered template instantiations, impacting code comprehension. Most templates are heavily specialised for different memory layouts, computations, etc., reducing orthogonality and code reuse. For example, CUTLASS contains different epilogues for GEMMs exploiting TCs and GEMMs using FPUs. Our GEMM API abstractions offer better separation of concerns and hence more reusability.

Like our GEMM API, CUTLASS contains both a BLAS-like interface, and an interface exposing all its flexibility. Launching a CUTLASS kernel using the latter requires more boilerplate compared to our approach in Listing 9, e.g. because CUTLASS users need to explicitly allocate a workspace that is used internally in CUTLASS.

NVIDIA’s CUBLASLT is a lightweight BLAS library dedicated to GEMM, with a more flexible API than CUBLAS. This flexibility comes in the form of support for more matrix layouts, input and compute data types, and algorithmic implementations. It is available on CUDA 10.1 or later.

Like our GEMM API, launching a kernel in CUBLASLT is also a two-step process. First, a “plan” must be created that determines the options for the GEMM computation. Second, this plan is used to launch one or more GEMM kernels.

CUBLASLT features concepts similar to CUTLASS, such as epilogues that post-process the resultant matrix and layouts describing how matrices are stored in memory. Each of these corresponds to an enumeration that lists the legal values, hence limiting flexibility. For example, it only includes bias and ReLU as possible epilogues, and offers no support for custom layouts such as diagonal matrices, or custom data types such as dual numbers. While CUBLASLT’s interface is an improvement over that of CUBLAS, its closed-source nature still results in limited extensibility.

BLIS is a framework that facilitates the instantiation of an entire BLAS library for new architectures, and hence has a larger scope than just GEMM [42]. It achieves this by rephrasing all BLAS operations in terms of a limited set of kernels. Their focus is on CPUs rather than GPUs, however. Similar to our GEMM kernel, BLIS contains a set components that can be reused for new BLAS-like operations. BLIS’s GEMM kernels offer support for more memory layouts and data types than traditional BLAS libraries. Nevertheless, BLIS’s flexibility is mainly aimed at developers of the BLIS library, instead of its users. For example, extending BLIS with support for complex numbers required significant effort, that warranted a separate paper describing its implementation [43].

Section 3 presented a tiling API that allows programmers to coordinate memory transfers to improve data locality. Automated tools based on polyhedral optimisation exist that can automatically generate tiled code from nested loops [44], [45], [46]. Basic approaches only reorder memory accesses, but more advanced ones can also exploit parallelism. For example, POLLY can exploit inter-tile parallelism using the OPENMP interface [47]. The framework by Baskaran et al. is even capable of automatically adding padding for shared memory accesses to reduce bank conflicts [48].

Recent work by Bondhugula uses the polyhedral utilities in MLIR to generate performant GEMM kernels [49]. His approach achieves a performance that is within 9% of state-of-the-art CPU GEMMs in BLIS and MKL. It still offers limited flexibility, however. Incorporating domain-specific optimisations, such as diagonal matrices, is significantly harder than in our approach, as it requires adaptations to the MLIR code base and/or TABLEGEN rules.

The DIESEL DSL (Domain-Specific Language) uses polyhedral techniques to compile high-level expressions to performant GPU kernels [50]. Bhaskaracharya et al. extend DIESEL with support for TCs and fused kernels that combine matrix multiplications with ReLU and bias [51]. Their work focuses only on Volta TCs, however, and does not address other forms of GEMM flexibility such as support for more complex data types. Additionally, only a limited number of element-wise operations are supported.

HALIDE, a DSL for image processing, was also extended with TC support by Sioutas et al. [52]. Their approach features a fixed kernel skeleton, similar to our template GEMM kernel. Their kernel also makes use of WMMA, and achieves performance results similar to ours. It still has limited flexibility, however. For example, it offers no support for complex data types, and only handles one combination of memory layouts for the  $A$  and  $B$  matrices, which necessitates explicit transposition kernels.

Note that BLIS, the polyhedral techniques (in so far as they support GPUs), DIESEL and HALIDE all involve statically compiled, statically typed PLs. For the general-purpose PLs, this by itself foregoes the productivity advantages of rapid-prototyping PLs such as Julia or Python. For the DSLs, it implies that code reuse across domains is limited. Those solutions also by construction do not consider flexibility beyond the data types, layouts, and operations typical for their domains. Our solution does not suffer from these drawbacks, yet obtains performance in the same ballpark.

## 7 AVAILABILITY

Our contributions are open source and available in the relevant GitHub repositories. Support for Tensor Cores using WMMA was merged into CUDA.jl, and is available in the latest stable version. The required adaptations to the Julia compiler were sent to the developers, and have been merged upstream. Our tiling and flexible GEMM APIs are bundled in one Julia package GEMMKERNELS.jl. This package is available at <https://github.com/thomasfaingnaert/GemmKernels.jl>, and can easily be installed using Julia's built-in package manager. It contains all instantiations of our API abstractions of all experiments and listings in this paper, ready for out-of-the-box re-use.

## 8 CONCLUSIONS AND FUTURE WORK

In this paper, we first presented tiling abstractions with which programmers can use tiling techniques, which are necessary to achieve high performance for many computations, at a high level of abstraction.

We then discussed a flexible GEMM API where the kernel consists of a set of orthogonal components. Each of these components corresponds to a set of Julia functions that can be specialised for different GEMM variants. We demonstrated the flexibility of this approach by instantiating the necessary components for 5 variants of GEMM computations: a normal mixed-precision GEMM, computations using diagonal matrices, computations exploiting operator fusion, GEMMs on complex and dual numbers, and a tensor contraction. We argued how specific features of the Julia compiler, such as multiple dispatch, type inference, and just-ahead-of-time compilation, allow for this flexibility without run-time overhead.

An experimental evaluation showed that the performance of our GEMM kernels written entirely in Julia is in the same ballpark as, and in some cases even exceeds, the state of the art in the manually tuned CUBLAS, CUTLASS, and CUTENSOR.

We presented two interfaces to use our flexible GEMM API: a fully-featured interface and a BLAS-like interface. The former exposes the full flexibility of our framework, the latter extends BLAS's GEMM with support for more data types such as dual numbers. We demonstrated our APIs for CUDA-enabled GPUs, but our abstractions are vendor-agnostic and can be ported to other GPU architectures.

In the future, we plan to port our framework to other GPUs such as those of AMD and Intel, and to add support for the `mma` family of instructions as well as data swizzling, as used in CUTLASS and CUBLAS, to improve performance.

At the moment, the matrix inputs to our kernels must be zero-padded such that their size is a multiple of the GEMM tile sizes. The engineering required to support arbitrary matrix dimensions, e.g. through the use of predicated memory accesses as done by CUTLASS, is also future work.

## ACKNOWLEDGEMENTS

This work was funded by the Research Foundation Flanders (Fonds voor Wetenschappelijk Onderzoek), grant number 3G051318.

## REFERENCES

- [1] BLAS contributors. (2017) BLAS (Basic Linear Algebra Subprograms).
- [2] A. Abdelfattah, S. Tomov, and J. Dongarra, "Towards half-precision computation for complex matrices: A case study for mixed precision solvers on GPUs," in *IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 2019, pp. 17–24.
- [3] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing GPU Tensor Cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.
- [4] T. Ichimura, K. Fujita, T. Yamaguchi, A. Naruse, J. C. Wells, T. C. Schultheiss, T. P. Straatsma, C. J. Zimmer, M. Martinasso, K. Nakajima, M. Hori, and L. Maddegedara, "A fast scalable implicit solver for nonlinear time-evolution earthquake city problem on low-ordered unstructured finite elements with artificial intelligence and transprecision computing," in *Proc. Int'l Conference for High Performance Computing, Networking, Storage, and Analysis*, 2018.
- [5] A. Haidar, H. Bayraktar, S. Tomov, and J. Dongarra, "Harnessing Tensor Cores FP16 arithmetic to accelerate linear solvers and HPC scientific applications," 2018, nVIDIA GPU Technology Conference.
- [6] V. Mehta, "Getting started with Tensor Cores in HPC," 2019, nVIDIA GPU Technology Conference.
- [7] D. Yan, W. Wang, and X. Chu, "Demystifying Tensor Cores to optimize half-precision matrix multiply," in *Proc. 34th IEEE International Parallel and Distributed Processing Symposium*, 2020.
- [8] NVIDIA. (2019, 6) Deep learning performance guide.
- [9] ———. (2020) NVIDIA V100.
- [10] P. Barham and M. Isard, "Machine learning systems are stuck in a rut," in *Proc. Workshop on Hot Topics in Operating Systems*, 2019, pp. 177—183.
- [11] N. Rink, A. Susungi, J. Castrillón, J. Stiller, and C. Tadonki, "CFDLang: High-level code generation for high-order methods in fluid dynamics," in *Real World Domain Specific Languages Workshop 2018*, 02 2018, pp. 1–10.
- [12] R. Poya, A. J. Gil, and R. Ortigosa, "A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics," *Computer Physics Communications*, vol. 216, pp. 35–52, 2017.
- [13] A. Auer, G. Baumgartner, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, G. Xiaoyang, R. Harrison, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Automatic code generation for many-body electronic structure methods: The tensor contraction engine," *Molecular Physics*, vol. 104, 01 2006.
- [14] P. Springer and P. Bientinesi, "The landscape of high-performance tensor contractions," in *Workshop on Batched, Reproducible, and Reduced Precision BLAS*, 2017.
- [15] T. Nelson, A. Rivera, P. Balaprakash, M. Hall, P. D. Hovland, E. Jessup, and B. Norris, "Generating efficient tensor contractions for GPUs," in *2015 44th International Conference on Parallel Processing*, 2015, pp. 969–978.
- [16] P. Springer and P. Bientinesi, "Design of a high-performance GEMM-like tensor-tensor multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 3, pp. 1–29, 2018.
- [17] E. D. Napoli, D. Fabregat-Traver, G. Quintana-Ortí, and P. Bientinesi, "Towards an efficient use of the BLAS library for multilinear tensor contractions," *Applied Mathematics and Computation*, vol. 235, pp. 454–468, 2014.

- [18] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc, "An input-adaptive and in-place approach to dense tensor-times-matrix multiply," in *Proc. Int'l Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.
- [19] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in *27th Int'l Symposium on Parallel and Distributed Processing*, 2013, pp. 813–824.
- [20] B. W. Bader and T. G. Kolda, "Algorithm 862: MATLAB tensor classes for fast algorithm prototyping," *ACM Transactions on Mathematical Software*, vol. 32, no. 4, pp. 635–653, December 2006.
- [21] J. Kim, A. Sukumaran-Rajam, V. Thumma, S. Krishnamoorthy, A. Panyala, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "A code generator for high-performance tensor contractions on GPUs," in *Proc. IEEE/ACM Int'l Symposium on Code Generation and Optimization*, 2019, p. 85–95.
- [22] D. A. Matthews, "High-performance tensor contraction without transposition," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C1–C24, 2018.
- [23] C. Psarras, L. Karlsson, J. Li, and P. Bientinesi, "The landscape of software for tensor computations," 2021.
- [24] NVIDIA. (2020) CUDA C++ programming guide.
- [25] Khronos Group. (2020) OpenCL: An open standard for parallel programming of heterogeneous systems.
- [26] JuliaLang.org. (2020) The Julia language.
- [27] ——. (2020) Julia micro-benchmarks.
- [28] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: Unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 827–841, 2019.
- [29] T. Besard, V. Churavy, A. Edelman, and B. De Sutter, "Rapid software prototyping for heterogeneous and distributed platforms," *Advances in Engineering Software*, vol. 132, pp. 29 – 46, 2019.
- [30] JuliaLang.org. (2020) The Julia language official documentation.
- [31] LLVM contributors. (2020) The LLVM compiler infrastructure project.
- [32] V. Churavy. (2020) GPUifyLoops.jl: Support for writing loop-based code that executes both on CPU and GPU.
- [33] G. Hinton, S. Sabour, and N. Frosst, "Matrix capsules with EM routing," in *International Conference on Learning Representations*, 2018.
- [34] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.
- [35] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, 2019, pp. 8024–8035.
- [36] E. Aprà, M. Klemm, and K. Kowalski, "Efficient implementation of many-body quantum chemical methods on the Intel® Xeon Phi coprocessor," in *Proc. Int'l Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 674–684.
- [37] W. Ma, S. Krishnamoorthy, O. Villa, and K. Kowalski, "GPU-Based Implementations of the Noniterative Regularized-CCSD(T) Corrections: Applications to Strongly Correlated Systems," *Journal of Chemical Theory and Computation*, vol. 7, no. 5, pp. 1316–1327, 2011.
- [38] P. Springer and P. Bientinesi, "Design of a high-performance GEMM-like tensor-tensor multiplication," 2016.
- [39] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in Julia," *arXiv:1607.07892 [cs.MS]*, 2016.
- [40] V. Churavy. (2020) KernelAbstractions.jl: Heterogeneous programming in Julia. [Online]. Available: <https://github.com/JuliaGPU/KernelAbstractions.jl>
- [41] NVIDIA. (2020) CUTLASS: CUDA templates for linear algebra subroutines.
- [42] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, Jun. 2015.
- [43] F. G. Van Zee, "Implementing high-performance complex matrix multiplication via the 1M method," *SIAM Journal on Scientific Computing*, vol. 42, no. 5, pp. C221–C244, 2020.
- [44] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proc. 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 101–113.
- [45] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–23, 2013.
- [46] P. Di, D. Ye, Y. Su, Y. Sui, and J. Xue, "Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on GPUs," in *Proc. 41st Int'l Conference on Parallel Processing*, 2012, pp. 350–359.
- [47] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.
- [48] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for GPGPUs," in *Proc. 22nd Annual Int'l Conference on Supercomputing*, 2008, pp. 225–234.
- [49] U. Bondhugula, "High performance code generation in MLIR: An early case study with GEMM," *preprint arXiv:2003.00532*, 2020.
- [50] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover, "Diesel: DSL for linear algebra and neural net computations on GPUs," in *Proc. 2nd ACM SIGPLAN Int'l Workshop on Machine Learning and Programming Languages*, 2018, pp. 42–51.
- [51] S. G. Bhaskaracharya, J. Demouth, and V. Grover, "Automatic kernel generation for Volta Tensor Cores," *arXiv preprint arXiv:2006.12645*, 2020.
- [52] S. Sioutas, S. Stuijk, T. Basten, L. Somers, and H. Corporaal, "Programming tensor cores from an image processing DSL," in *Proc. 23th Int'l Workshop on Software and Compilers for Embedded Systems*, 2020, pp. 36–41.



**Thomas Faingnaert** is a PhD student at Ghent University in the Computer Systems Lab. He obtained his MSc degree in Computer Science Engineering from Ghent University's Faculty of Engineering and Architecture in 2020. His research focuses on software protection, and high-level abstractions for GPU programming in Julia.



**Tim Besard** is a software engineer at Julia Computing. He obtained his MSc in Computer Engineering from University College Ghent in 2011, and his PhD in Computer Science Engineering from Ghent University in 2019. He is currently the lead maintainer of several GPU back-ends for the Julia programming language.



**Bjorn De Sutter** is associate professor at Ghent University in the Computer Systems Lab. He obtained his MSc and PhD degrees in Computer Science from Ghent University's Faculty of Engineering in 1997 and 2002. His research focuses on the use of compiler techniques to aid programmers with non-functional aspects of their software, such as performance, code size, reliability, and security.

# FP8-LM: Training FP8 Large Language Models

Houwen Peng \* Kan Wu \* Yixuan Wei \*

Guoshuai Zhao Yuxiang Yang Ze Liu Yifan Xiong Ziyue Yang

Bolin Ni Jingcheng Hu Ruihang Li Miaosen Zhang Chen Li Jia Ning Ruizhe Wang Zheng Zhang  
Shuguang Liu Joe Chau Han Hu † Peng Cheng †

**Microsoft Azure and Microsoft Research**

## Abstract

In this paper, we explore FP8 low-bit data formats for efficient training of large language models (LLMs). Our key insight is that most variables, such as gradients and optimizer states, in LLM training can employ low-precision data formats without compromising model accuracy and requiring no changes to hyper-parameters. Specifically, we propose a new FP8 automatic mixed-precision framework for training LLMs. This framework offers three levels of FP8 utilization to streamline mixed-precision and distributed parallel training for LLMs. It gradually incorporates 8-bit gradients, optimizer states, and distributed learning in an incremental manner. Experiment results show that, during the training of GPT-175B model on H100 GPU platform, our FP8 mixed-precision training framework not only achieved a remarkable 42% reduction in real memory usage but also ran 64% faster than the widely adopted BF16 framework (i.e., Megatron-LM), surpassing the speed of Nvidia Transformer Engine by 17%. This largely reduces the training costs for large foundation models. Furthermore, our FP8 mixed-precision training methodology is generic. It can be seamlessly applied to other tasks such as LLM instruction tuning and reinforcement learning with human feedback, offering savings in fine-tuning expenses. Our FP8 low-precision training framework is open-sourced at [aka.ms/MS.AMP](https://aka.ms/MS.AMP).

## 1 Introduction

Large language models (LLMs) (Brown et al., 2020; Smith et al., 2022; Chowdhery et al., 2022; Zhang et al., 2022) have demonstrated unprecedented capabilities in language comprehension and generation, leading to breakthroughs in reasoning, math, science, and many other tasks (OpenAI, 2023; Anil et al., 2023). However, training LLMs is extremely costly. For example, PaLM takes 6,144 TPUs v4 chips to train a 540B model, while GPT-3 175B consumes several thousand petaflop/s-days of compute for pre-training (Chowdhery et al., 2022; Brown et al., 2020). This motivates the needs of reducing the training costs of LLMs, especially for the scaling of next-generation super-intelligent models.

Low-precision training is one of the most promising directions to reduce the costs, as it can provide high speed, small memory footprint, and low communication overhead. Most existing training systems, e.g., Megatron-LM (Shoeybi et al., 2019), MetaSeq (Zhang et al., 2022), and Colossal-AI (Li et al., 2023a), train LLMs with either FP32 full-precision or FP16/BF16 mixed-precision by default. This is not essential, however,

Contributions for all the authors can be found in Section 5.

\* equal work † contact: {hanhu | pengc}@microsoft.com

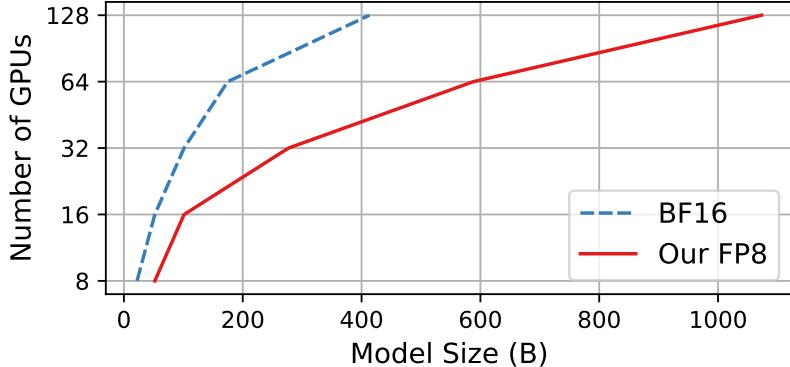


Figure 1: An analysis of comparing the maximum model sizes attainable through the utilization of either the prevalent BF16 or our FP8 mixed-precision training approach on a cluster of Nvidia H100 GPUs with 80G memory.

to achieve full accuracy for large models. With the release of Nvidia H100 GPU, FP8 is becoming the next-generation datatype for low-precision representation ([Nvidia, 2022a](#); [Micikevicius et al., 2022](#)). Theoretically, FP8 can achieve 2x speed-up, 50% - 75% memory cost savings, and 50% - 75% communication savings compared with current 16-bit and 32-bit floating point mixed-precision training, which is very promising for scaling-up next-generation foundation models.

Unfortunately, the current support for FP8 training is rare and limited. The only usable framework is the Nvidia Transformer Engine (TE) ([Nvidia, 2022b](#)), but it applies FP8 solely for GEMM computation and still retains master weights and gradients using high precision, e.g., FP16 or FP32. As a result, the end-to-end speed-up, memory and communication cost savings are very limited, which does not fully unveil the power of FP8. To address this issue, we propose an extremely optimized FP8 mixed-precision framework for LLM training. The core idea is to infiltrate FP8 compute, storage, and communication into the whole progress of large model training, making the forward and backward pass all used the low-precision FP8, thus largely reducing system workloads compared to previous frameworks ([Micikevicius et al., 2017](#); [Nvidia, 2022b](#); [Micikevicius et al., 2022](#)). Specifically, we design three optimization levels that utilize FP8 to streamline mixed-precision and distributed training. The three levels gradually incorporate 8-bit collective communication, optimizer, and distributed parallel training in an incremental manner. The higher optimization level indicates using more FP8 during LLM training. Moreover, for large-scale training, such as GPT-175B trained on thousand of GPUs, our framework provides FP8 low-bit parallelism, including tensor, pipeline, and sequence parallelism, paving the way to next-generation low-precision parallel training.

Training LLMs with FP8 is non-trivial. The challenges stem from issues such as data underflow or overflow, coupled with quantization errors arising from the narrower dynamic range and reduced precision inherent in FP8 data formats. These challenges cause numerical instabilities and irreversible divergences throughout the training process. To tackle them, we propose two techniques: *precision decoupling* and *automatic scaling* for preventing the loss of critical information. The former one involves decoupling the influence of data precision on parameters such as weights, gradients, optimizer states, and assigning reduced precision to components that are not precision sensitive. The latter one is to preserve gradient values within the representation range of FP8 data formats through the dynamic adjustment of tensor scaling factors, thereby alleviating underflow and overflow occurrences during all-reduce communication.

To validate the proposed FP8 low-precision framework, we apply it to GPT-style model training, encompassing both pre-training and supervised fine-tuning (SFT). The experimental results demonstrate the effectiveness of our FP8 methodology, yielding substantial benefits including a 27% to 42% reduction in real memory usage (e.g., 27% reduction for GPT-7B while 42% for GPT-175B) and a notable 63% to 65% decrease in weight gradient communication overhead compared to the prevalent BF16 mixed-precision training approach. Without changes to any hyper-parameters, such as learning rate and weight decay, the models trained using FP8 exhibit performance equivalency to those employing BF16 high precision, both in pre-training and

downstream tasks. It is noteworthy that during the training of GPT-175B model, our FP8 mix-precision framework reduces training time by 17% compared to TE ([Nvidia, 2022b](#)), while consuming 21% less memory on H100 GPU platform. More importantly, the reduction in costs achieved through the utilization of low-precision FP8 can be further increased, as the scale of models continues to expand, which is presented in Fig. 1.

For fine-tuning, we employ FP8 mixed-precision for instruction tuning and reinforcement learning with human feedback (RLHF) to better align pre-trained LLMs with end tasks and user preferences. Specifically, we fine-tune pre-trained models on publicly user-shared instruction-following data ([ShareGPT, 2023](#)). The models tuned with our FP8 mixed-precision demonstrate comparable performance to those utilizing the half-precision BF16 ([Zheng et al., 2023](#)) on the AlpacaEval ([Li et al., 2023b](#)) and MT-Bench ([Zheng et al., 2023](#)) benchmarks, while achieving 27% improvements in training speed. Moreover, FP8 mixed-precision exhibits considerable potentials in RLHF, a process that necessitates loading multiple models during training. Through the utilization of FP8 in training, the prevalent RLHF framework AlpacaFarm ([Dubois et al., 2023](#)) can yield a 46% reduction in model weights and a 62% reduction in optimizer states' memory consumption. This further demonstrates the versatility and adaptability of our FP8 low-precision training framework.

We are making the following contributions to drive the design of next-generation FP8 low-precision training for LLMs.

- A new FP8 mixed-precision training framework. It unlocks 8-bit weights, gradients, optimizer, and distributed training gradually in an add-on fashion, which is convenient in use. This 8-bit framework can be used as a simple drop-in replacement for existing 16/32-bit mixed-precision counterparts, without requiring any changes to the hyper-parameters and training receipts. Additionally, we provide a Pytorch implementation that enables 8-bit low-precision training in a few lines of code.
- A new family of GPT-style models trained with FP8. We apply the proposed FP8 scheme to GPT pre-training and fine-tuning (*i.e.*, SFT and RLHF), and demonstrate its potentials on a variety of model scales ranging from 7B to 175B parameters. We equip prevalent parallel computation paradigms with FP8 supports, including tensor, pipeline, and sequence parallelisms, enabling the utilization of FP8 to train large foundation models. We open-source the first FP8 GPT training codebase based upon Megatron-LM ([Shoeybi et al., 2019](#)) implementation.

We expect the release of our FP8 framework will establish a new paradigm for next-generation low-precision training system dedicated to large foundation models.

## 2 FP8 LLMs

Mixed-precision ([Micikevicius et al., 2017](#)) has been widely used in LLM training to improve compute and memory efficiency. The most popular mixed-precision schemes are FP16-FP32 and BF16-FP32. Because of the restricted numerical range of FP16, FP16-FP32 scheme has been known instabilities for training large models ([Rae et al., 2021; Zeng et al., 2022](#)). Consequently, the community now commonly adopts BF16-FP32 for training LLMs, such as Megatron-Turing NLG-530B ([Smith et al., 2022](#)), Bloom-175B ([Scao et al., 2022](#)) and Gopher ([Rae et al., 2021](#)). The underlying reason is that BF16 has a wide dynamic range to maintain numerical stability while matching the performance of the full-precision FP32. Moreover, BF16 employs half the number of bits as compared to FP32, thus reducing considerable memory footprints while improving compute efficiency.

FP8 is a natural evolution from 16-bit data formats to further reducing computing costs. However, training LLMs with reduced-precision FP8 poses new challenges. The dynamic range and representation precision of FP8<sup>1</sup> are much lower than BF16 and FP16, which inevitably induces more training collapses, such as loss spikes or even NaNs. To address the issues, tensor scaling techniques are proposed ([Sun et al., 2019; Micikevicius et al., 2022](#)). The core idea is multiplying higher precision values with a scaling factor prior to their casting to FP8 in order to move them into a range that better overlaps with the representable range of

<sup>1</sup>The details of FP8 data formats are presented in Appendix A.1.

a corresponding FP8 format<sup>2</sup> (Micikevicius et al., 2022). Such a per-tensor scaling technique reduces data quantization errors while improving numerical stability and accuracy, thus enabling the utilization of the lower-precision FP8 for training large models.

Unfortunately, the current support for FP8 low-precision training is restricted. Nvidia TE (Nvidia, 2022b) only supports FP8 compute for linear layers in Transformer (Vaswani et al., 2017), while leaving all other operations, such as weight update and gradient synchronization, still using higher precision. In this work, we present an extremely optimized FP8 mixed-precision strategy for LLM training. The new FP8 optimization includes three key perspectives: FP8 communication, FP8 optimizer, and FP8 distributed training. By integrating these aspects, the training of LLMs such as the 175B GPT-3 model can fully harness the advantages of FP8 low-precision and improve training efficiency.

## 2.1 FP8 Gradient and All-Reduce Communication

Existing mixed-precision training methodologies (Micikevicius et al., 2017; Nvidia, 2022b) typically employ 16-bit or 32-bit datatype for the computation and storage of gradients, resulting in a high bandwidth requirement for collective communication throughout the training process. We found that directly applying FP8 to gradients leads to a decrease in accuracy. The fundamental issue lies in the underflow and overflow problems arising from the low-bit all-reduce operation. Specifically, there are two standard methods aggregating gradients across GPUs during all-reduce: *pre-scaling* and *post-scaling*. Pre-scaling divides the gradient  $g_i$  calculated on the  $i$ -th GPU by the total number of GPUs (*i.e.*,  $N$ ) before being summed, which is formulated as:

$$g = g_1/N + g_2/N + \dots + g_N/N. \quad (1)$$

When  $N$  is large, this division can cause data underflow, especially for FP8 low-precision representation of gradients. To mitigate this issue, post-scaling performs the gradient summation first, followed by the division scaling during the gradient collection process:

$$g = (g_1 + g_2 + \dots + g_N)/N. \quad (2)$$

This post-scaling approach keeps the gradients close to the maximum value of the FP8 datatype, effectively alleviating the underflow issue. However, this approach encounters overflow issues when aggregating gradients.

In contrast, we propose an *automatic scaling* technique to resolve both the underflow and overflow issues in the pre-scaling and post-scaling approaches. To be specific, we introduce an auto-scaling factor  $\mu$ , that changes on the fly during the training, to reduce the occurrences of overflow and underflow in gradients:

$$g'_i = \mu \cdot g_i. \quad (3)$$

A statistical analysis is conducted on the gradient values of  $g'_i$ , with the objective of quantifying the proportion of values that attains the maximum feasible value within the FP8 representation range. If the ratio of the maximum value exceeds a specified threshold, *i.e.*, 0.001%,  $\mu$  is set to 1/2 in the subsequent training step, thereby mitigating the risk of overflow. Conversely, when the ratio consistently remains the threshold, we opt to exponentially increase  $\mu$  to 2 over the span of 1,000 training steps, thereby effectively mitigating the risk of underflow occurrences.

Another key obstacle of FP8 collective communication lies in devising an effective strategy to manage the tensor-wise scaling factors that are associated with each gradient tensor. The current NCCL implementation (Nvidia, 2020) lacks the capability of performing all-reduce operation considering the additional tensor-wise scaling factors. Meanwhile, efficient implementation is also very challenging, especially considering that the NCCL gradient summation operates at sub-tensor level. This complexity increases significantly when incorporating updates for tensor-wise scaling factors. To overcome this issue, we propose a new mechanism that scales FP8 gradients across GPUs using a single shared scalar. To be specific, let  $(g'_i, s'_i)$  denote a scaling tensor which stores the weight gradient in the  $i$ -th GPU, where  $g'_i$  is a FP8 tensor and  $s'_i$  is the corresponding scaling factor. The actual weight gradient is  $g'_i / s'_i$ . Prior to the all-reduce operation over gradient tensors,

---

<sup>2</sup>The details of FP8 tensor scaling are introduced in Appendix A.2.

we first gather the scaling factors  $s'_i$  of each gradient tensor on all GPUs and calculate the global minimum scaling factor  $s'_g$  as:

$$s'_g = \min(s'_1, s'_2, \dots, s'_N), \quad (4)$$

where the global minimum scaling factor  $s'_g$  is shared across GPUs. We use this shared scaling factor  $s'_g$  to unify the rescaling of the gradient tensors across GPUs. In this way, all gradient tensors associated with the same weight use the same shared scaling factor to quantize the tensors into FP8 format on all GPUs:

$$g''_i = \text{FP8}(s'_g \cdot (g'_i / s'_i)). \quad (5)$$

This approach reduces communication overhead by transmitting only a single scalar  $s'_g$ , making the additional synchronization step highly efficient. As the input tensors share the same scaling factor, it eliminates the need of considering all-reduce the scaling factors in parallel and allows for standard NCCL all-reduce operation to be performed. The final collected gradient is obtained as follows:

$$g = g''_1 + g''_2 + \dots + g''_N, \quad s = N \cdot s'_g, \quad (6)$$

where  $g$  is the final aggregated gradient and  $s$  is the corresponding scaling factor. Rescaling the scaling factor for the summed gradient  $g$  is equivalent to dividing  $g$  by  $N$  in theory. By implementing the aforementioned dual strategies of distributed and automated scaling, we can successfully realize FP8 low-bit gradient communication while preserving model accuracy. Furthermore, this approach entails storing gradients in FP8 and conducting communication in FP8 as well, thereby yielding reductions in GPU memory usage and communication bandwidth consumption.

## 2.2 FP8 Optimizer

In the training of LLMs, Adam and its variants ([Kingma and Ba, 2015](#); [Loshchilov and Hutter, 2018](#)) are the most frequently-used optimization methods, that maintain copies of model weights, gradients, first-order and second-order gradient moments for model updates. Mixed-precision training ([Micikevicius et al., 2017](#)) with Adam optimizer typically stores master weights, gradients and gradient moments in 32-bit float format for numerical stability ([Shoeybi et al., 2019](#); [Rajbhandari et al., 2020](#); [Zhang et al., 2022](#); [Scao et al., 2022](#)). Consequently, the Adam optimizer consumes 16 bytes of memory per parameter during training:

$$\underbrace{4}_{\text{master weights}} + \underbrace{4}_{\text{gradients}} + \underbrace{4 + 4}_{\text{Adam states}} = 16 \text{ bytes.} \quad (7)$$

When model size is large, the memory consumption of the variables in Adam will become a bottleneck. Previous work ([Rae et al., 2021](#); [Zeng et al., 2022](#); [Liu et al., 2022](#)) has revealed that reducing precision of the variables in optimizer to 16-bit leads to accuracy degradation when training billion-scale models<sup>3</sup>. This prompts an evaluation of which variables in the optimizer should be allocated high precision and which can be accommodated with low-precision.

To clarify, we decouple the influence of data precision on the variables in the optimizer and investigate which one can be assigned lower precision, *i.e.*, *precision decoupling*. We find a guiding principle: the gradient statistics can use lower precision, while the master weights necessitate high precision. More concretely, the first-order gradient moment can tolerate a high quantization error and can be assigned with low-precision FP8, while the second-order moment requires a higher precision, as analyzed in Sec. 3.3. This stems from the fact that, during model updates in Adam, the direction of the gradient holds greater significance than its magnitude. FP8 with tensor scaling can effectively preserve the distribution of the first-order moment as the high-precision tensor, though it introduces precision degradation to some extend. Calculating the square of gradients for the second-order gradient moment might lead to data underflow due to the typically small gradient values. Therefore, allocating a 16-bit higher precision is necessary to preserve numerical accuracy.

On the other hand, we find that it is crucial to keep the master weights using high precision. The underlying reason is that weight updates can sometimes become extremely small or large during training, higher

---

<sup>3</sup>BF16 lacks the precision needed for accuracy, while FP16 has a restricted dynamic range. Given these challenges, prevalent mixed-precision training methodologies rely on utilizing FP32 full-precision for master weights, gradients, and gradient moments.

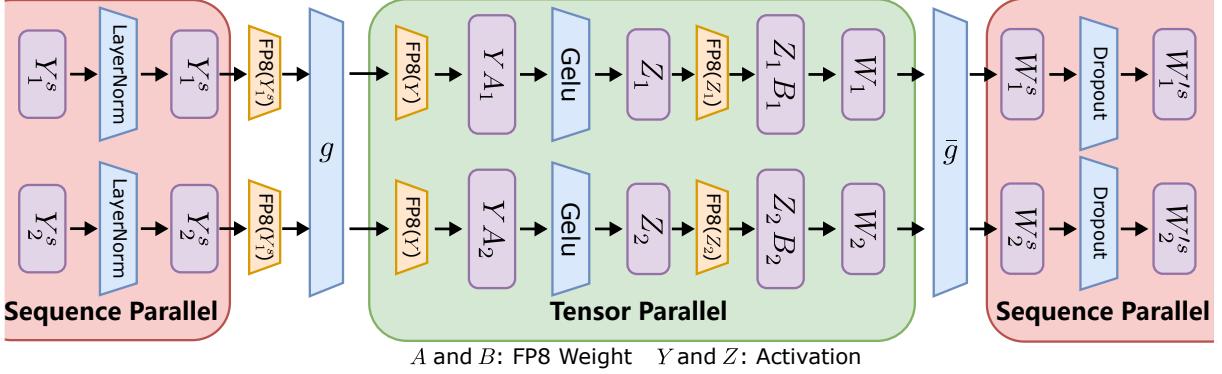


Figure 2: Transformer layer with FP8 tensor and sequence parallelism. The FP8 low-bit operation is highlighted with orange.  $g$  is all-gather in forward pass and reduce-scatter in backward pass, while  $\bar{g}$  is reduce-scatter in forward pass and all-gather in backward pass. The gather-reduce operation  $g$  between sequence parallel and tensor parallel is executed utilizing FP8 low-precision activation, thus saving half communication costs.

precision for the master weights helps prevent loss of information when updating weights, ensuring more stable and accurate training. In the implementation, the master weights have two viable options: utilizing either FP32 full-precision or FP16 with tensor scaling. FP16 with tensor scaling offers the advantage of conserving memory without compromising accuracy. Consequently, our default choice is to employ FP16 with tensor scaling for storing master weights in the optimizer. Our FP8 mixed-precision optimizer consumes 6 bytes of memory per parameter during training:

$$\underbrace{2}_{\text{master weights}} + \underbrace{1}_{\text{gradients}} + \underbrace{1 + 2}_{\text{Adam states}} = 6 \text{ bytes.} \quad (8)$$

This new low-bit optimizer reduces memory footprints by 2.6x in comparison to the previous solution, as exemplified in Eq. (7). Noteworthily, this is the first FP8 optimizer for LLM training. The experiments in Sec. 3.2 show that FP8 optimizer can preserve model accuracy at various scales, ranging from 125M to 175B parameters.

### 2.3 FP8 Distributed Parallel Training

Training LLMs like GPT-3 requires distributed learning strategies for parallelizing across GPUs. The frequently-used strategies include data parallelism, tensor parallelism, pipeline parallelism, and sequence parallelism. Each parallelism has its own merits and has been used in a complementary fashion in existing systems (Smith et al., 2022; Shoeybi et al., 2019; Zhang et al., 2022; Scao et al., 2022; Li et al., 2023a). For FP8 supports of these strategies, neither data parallelism nor pipeline parallelism necessitates any specific modifications, because they do not involve additional FP8 compute and communication when splitting data batches or model layers into segments across devices.

Tensor parallelism partitions individual layers of a model across multiple devices, such that the shards of weight, gradient and activation tensors are placed on separate GPUs, instead of a single one. To equip tensor parallelism with FP8, we convert the sharded weight and activation tensors to FP8 format for linear layer computation, enabling the forward compute and backward gradient collective communication all using FP8. On the other hand, sequence parallelism splits input sequences into multiple chunks and the sub-sequences are fed to different devices to save activation memory. As shown in Fig. 2, sequence and tensor parallelism are performed in parallel to different parts of a Transformer model to make the best use of the available memory and improve training efficiency. There is a converter  $g$  between sequence and tensor parallel regions to all-gather sequence partitions in the forward pass (or reduce-scatter tensor segments in the backward pass). We add an FP8 datatype conversion prior to  $g$ , such that the all-gather (or reduce-scatter) operation uses FP8 low-bit activation to save communication cost across GPUs.

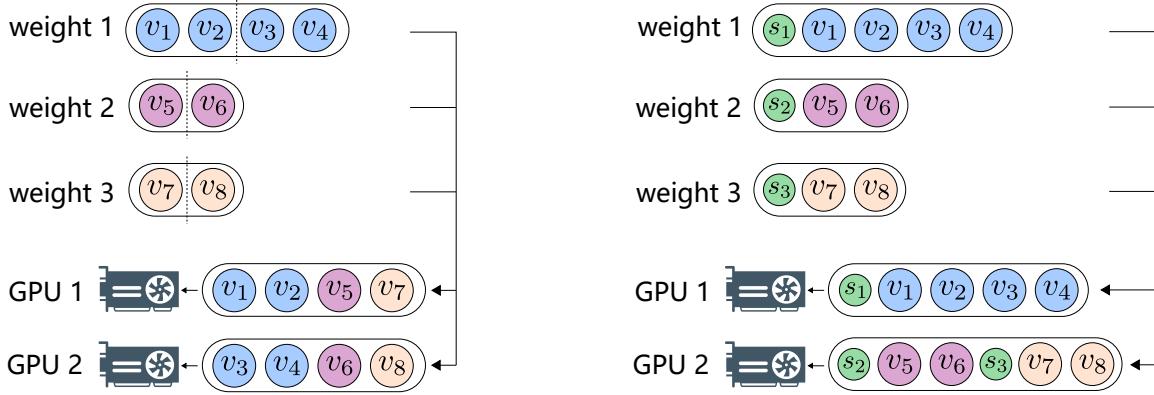


Figure 3: ZeRO tensor partitioning with and without scaling factors. Left: the original high-precision ZeRO method, which splits a single tensor into multiple partitions and distributes them to different devices. Right: the proposed FP8 ZeRO, which distributes each tensor in its entirety across devices while taking tensor scaling into account.

In addition, Zero Redundancy Optimizer (ZeRO) (Rajbhandari et al., 2020) is another frequently-used distributed learning technique in large model training. The core idea of ZeRO is to shade model states over devices, such that each device only hold a fraction of data (e.g., master weights, gradients, and optimizer states) required for a training step. To reduce memory consumption, ZeRO method generally splits a single tensor into multiple partitions and distributes them to different devices. Directly applying FP8 to ZeRO is infeasible, because it is difficult to handle the scaling factors associated with the FP8 partitions. The per-tensor scaling factors should be distributed along with FP8 partitions. To address this issue, we implement a new FP8 distribution scheme that distributes each tensor as a whole across devices, rather than partitioning it into multiple sub-tensors as in ZeRO (Rajbhandari et al., 2020). The distribution of FP8 tensors is processed in a greedy manner, as outlined in Alg. 1. Specifically, our method first sorts the tensors of model states according to their sizes, and then distributes the tensors to different GPUs based upon the remaining memory size of each GPU. The distribution follows the principle that the GPUs with larger remaining memory get a higher priority in receiving new distributed tensors. In this way, the tensor scaling factors can be distributed along with the tensors smoothly, while reducing communication and compute complexity. Figure 3 presents a visual illustration of the difference in ZeRO tensor partitioning between scenarios with and without scaling factors.

### 3 Experiment

In this section, we assess the effectiveness of the proposed FP8 mixed-precision training approach on GPT-style LLMs, including a wide range of model scales, from 125 million to 175 billion parameters. For performance

---

#### Algorithm 1 Greedy Distribution Algorithm for ZeRO

---

**Input:** FP8 tensors with their corresponding scaling factors:  $T = \{(s_1, t_1), (s_2, t_2), \dots, (s_n, t_n)\}$ , where  $s$  denotes scaling factors while  $t$  represents 8-bit tensors. The size of each tensor:  $C = \{c_1, c_2, \dots, c_n\}$ .  
**Output:** Partitions representing scaling tensors assigned to each GPU.  
 1: Sort  $T$  in descending order of their sizes to get  $T' = \{(s'_1, t'_1), (s'_2, t'_2), \dots, (s'_n, t'_n)\}$  and  $C' = \{c'_1, c'_2, \dots, c'_n\}$ , where  $c'_1 \geq c'_2 \geq \dots \geq c'_n$ .  
 2: Initialize memory usage  $u_j = 0$  and partition  $p_j = \emptyset$  for each GPU  $G_j$ .  
 3: **for**  $i = 1$  to  $n$  **do**  
 4:      $j \leftarrow \arg \min_j u_j$  ▷ Find the GPU  $j \in [1, m]$  with the least memory usage.  
 5:      $p_j \leftarrow p_j \cup \{(s'_i, t'_i)\}$  ▷ Assign  $(s'_i, t'_i)$  to  $G_j$ .  
 6:      $u_j \leftarrow u_j + c'_i$  ▷ Update the memory usage of  $G_j$ .  
 7: **end for**  
 8: **return** Partitions  $P = \{p_1, p_2, \dots, p_m\}$

---

params	dimension	<i>n</i> heads	<i>n</i> layers	TP	PP	SP	learning rate	batch size	<i>n</i> tokens
125M	768	12	12	1	1	✓	$6.0e^{-4}$	1M	100B
7B	4096	32	32	1	1	✓	$3.0e^{-4}$	4M	100B
13B	5120	40	40	2	1	✓	$3.0e^{-4}$	4M	100B
175B	12288	96	96	8	4	✓	$3.0e^{-5}$	1M	5B

Table 1: Model sizes, architectures, and training hyper-parameters. TP, PP, and SP indicate tensor, pipeline, and sequence parallelism, respectively. To mitigate carbon emissions and save cost, we restrict the training of the 175B model to a dataset comprising only 5B tokens, which has proven to be sufficient for evaluating system performance.

ablation, we compare GPT models trained with FP8 against those trained with half-precision BF16 and full-precision FP32. For generality evaluation, we conduct experiments encompassing both FP8 low-bit pre-training and fine-tuning, considering instruction tuning and human preference alignment.

### 3.1 Experimental Setup

#### 3.1.1 Training Dataset

Our pre-training data is constructed using open-sourced language collections from several sources, including CommonCrawl<sup>4</sup>, The Pile (Gao et al., 2020), C4 (Raffel et al., 2020), OpenWebText (Radford et al., 2019; Gokaslan and Cohen, 2019), CC-NEWS (Liu et al., 2019), CC-Stories (Trinh and Le, 2018), Redpajama (Redpajama, 2023), and Wikipedia<sup>5</sup>. We apply fuzzy deduplication (Lee et al., 2022) across CommonCrawl snapshots to enhance data quality. Tab. 10 in Appendix A.3 provides details of our pre-training data, including information such as the number of tokens from each source and associated sampling weights. For a more comprehensive understanding of the data and its cleaning pipeline, readers are encouraged to refer to Appendix A.3.

Moreover, for instruction tuning, we follow the same settings as Vicuna-v1.1 (VicunaTeam, 2023), which uses a publicly user-shared instruction following data (ShareGPT, 2023). For reinforcement learning with human feedback, the training data we used is a combination of the Anthropic’s Helpful and Harmless dataset (Bai et al., 2022) and Open-Assistant dataset (Köpf et al., 2023). The training framework and associated configurations align with the publicly available AlpacaFarm (Dubois et al., 2023).

#### 3.1.2 Model Configuration

The model architecture we used is a decoder-only Transformer (Brown et al., 2020), which has been widely-used in recent generative LLMs like PaLM (Chowdhery et al., 2022), OPT (Zhang et al., 2022), and LLaMA (Touvron et al., 2023). In addition to the base architecture, we integrate several modifications proposed recently to improve model efficiency and effectiveness. 1) *Rotary Positional Embedding*: Drawing inspiration from recent successful experiments (Black et al., 2022; Touvron et al., 2023), we incorporate rotary positional embeddings (RoPE) (Su et al., 2021) into our approach. This addition enables us to capture both absolute and relative positions information, enhancing performance especially when extrapolating to larger context windows. 2) *Flash Attention*: The standard attention implementation is bottlenecked by memory access (Ivanov et al., 2021). Flash Attention (Dao et al., 2022) proposed an IO-aware exact attention algorithm which uses tiling to reduce the amount of HBM accesses, achieving substantial acceleration.

We train the models using the proposed FP8 optimizer, which is built upon Adam (Kingma and Ba, 2015) with decoupled weight decay (Loshchilov and Hutter, 2018), following the common practise with the decay rates  $\beta_1 = 0.9$ ,  $\beta_2 = 0.95$ , and weight decay = 0.1. The learning rate schedule is cosine-like, and the final learning rate is 10% of the maximal learning rate. We train the models for 100B tokens in total with a batch size of 4M tokens, and the input sequence length is set to 2048. The model warm-up is conducted for 1,000

<sup>4</sup><https://commoncrawl.org>

<sup>5</sup><https://wikipedia.org>

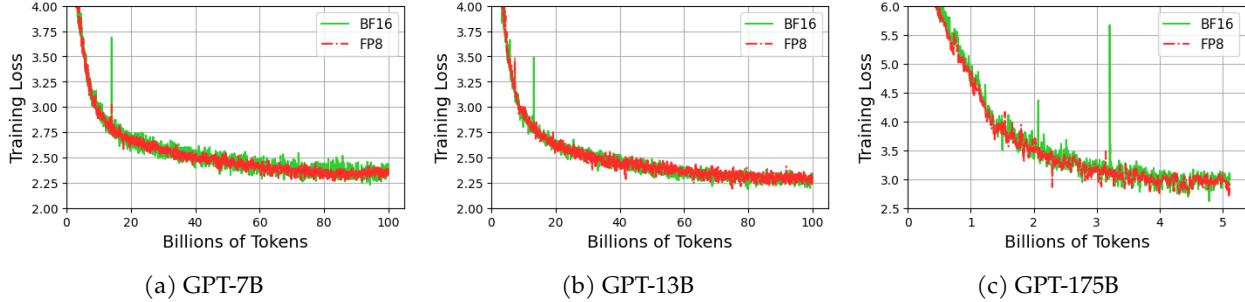


Figure 4: A comparison between FP8 and BF16: Analyzing the training loss of GPT models with the parameters ranging from 7 billion to 175 billion.

	HS	Lambada	BoolQ	PIQA	COPA	Winogrande	Arc-C	Arc-E	ObQA	Avg
<i>GPT-7B model zero-shot performance</i>										
<b>BF16</b>	61.3	61.4	61.2	75.0	79.0	58.5	32.9	59.7	36.4	58.4
<b>FP8</b>	60.0	61.8	62.0	74.2	78.0	59.8	32.9	58.7	34.6	58.0
<i>GPT-13B model zero-shot performance</i>										
<b>BF16</b>	64.8	64.9	63.4	75.9	82.0	61.0	35.2	61.5	40.6	61.0
<b>FP8</b>	64.1	63.4	63.9	76.2	81.0	61.6	34.9	61.3	36.8	60.4

Table 2: Zero-shot performance on downstream tasks. The models are trained with either the standard BF16 mixed-precision scheme (Shoeybi et al., 2019) or the proposed FP8 low-precision scheme.

iterations. Tab. 1 presents the details of model configurations and the corresponding training settings. The training is conducted on Azure NDv5 H100 GPU platform ([Microsoft, 2023](#)).

### 3.2 Main Results

### 3.2.1 Model Performance

We first compare the performance of models trained using FP8 mixed-precision with those trained using BF16. In Fig. 4, the pre-training loss over tokens is displayed for GPT models of 7B, 13B, and 175B parameters. The training configurations and hyper-parameters remain consistent across models trained with FP8 and BF16. The only difference lies in the mixed-precision schemes utilized. As shown in Fig. 4, the loss curves almost overlap with each other. The results unequivocally demonstrate that the proposed FP8 mixed-precision scheme can achieve equivalent performance to the prevalent higher-precision BF16 scheme (Shoeybi et al., 2019; Rae et al., 2021; Hoffmann et al., 2022) across a diverse array of model scales. Also, we evaluate the pre-trained models on a wide range of downstream tasks, including HellaSwag (HS) (Zellers et al., 2019), Lambada (Paperno et al., 2016) BoolQ (Clark et al., 2019), PIQA (Bisk et al., 2020), COPA (Roemmele et al., 2011), Winogrande (Sakaguchi et al., 2021), Arc (Clark et al., 2018), and OpenbookQA (ObQA) (Mihaylov et al., 2018). As reported in Tab. 2, the FP8 pre-trained models exhibit comparable zero-shot performance in comparison to their BF16 counterparts. This result provides further validation that models pre-trained with FP8 low-precision maintain both accuracy and intrinsic in-context learning capabilities at a level comparable to their high-precision counterparts.

Furthermore, we leverage the proposed FP8 mixed-precision approach for fine-tuning LLMs in instruction following. For a fair comparison, we follow the same instruction tuning settings as Vicuna-v1.1 ([VicunaTeam, 2023](#)), which adopts the open-sourced LLaMA-7B ([Touvron et al., 2023](#)) as the base model for fine-tuning. Fig. 5 presents the fine-tuning loss, where the curves corresponding to BF16 and FP8 display a notable degree of overlap. Meanwhile, the win-rate of our FP8 fine-tuned models against Davinci-003 ([OpenAI, 2022](#)) is also comparable to that of Vicuna-v1.1, which is fine-tuned using BF16 half-precision, as reported in Tab. 3. This indicates that our FP8 low-bit training scheme is versatile, as it is applicable not only to pre-training phase but also to downstream fine-tuning tasks.

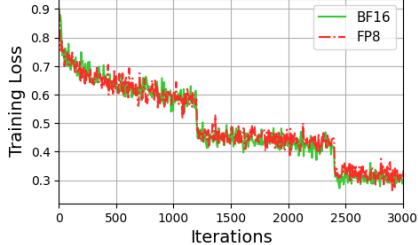


Figure 5: SFT training loss.

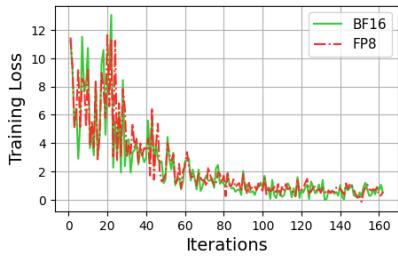


Figure 6: RLHF training loss.

In addition, we further apply the proposed FP8 mixed-precision scheme to reinforcement learning from human feedback (RLHF), a more complex process to align LLMs with user preferences. Following the same training setting as AlpacaFarm (Dubois et al., 2023), a recent RL framework for LLM alignment, we optimize policy models with PPO algorithm (Schulman et al., 2017). The sole difference lies in the choice of mixed-precision training schemes, *i.e.*, BF16 *v.s.* FP8. From the results reported in Fig. 6 and Tab. 4, we observe a notable reduction in memory utilization, for instance, a 31.8% memory reduction concerning model weights and a 62.5% reduction concerning optimizer states. Consequently, it can be inferred that FP8 is capable of replicating the BF16 mixed-precision for RLHF training. This underscores the broader applicability and versatility of our FP8 low-bit training solution.

### 3.2.2 System Performance

In this section, we evaluate system-level performance of FP8 mixed-precision, considering communication efficiency, memory utilization, and the overall speed, with an emphasis on cost savings. Our method employs 8-bit gradients for all-reduce collective communication among GPUs. Theoretically, this results in a 75% reduction in communication costs when compared to the mainstream 32-bit scheme (Despite BF16 mixed-precision computing gradients using 16-bit precision, it still employs 32-bit precision for all-reduce communication (Shoeybi et al., 2019)). Due to the impact of system transmission loss, the observed practical reduction during GPT model training falls within the range of 63% to 65%, as indicated in Table 5. Furthermore, it is worth noting that the recent Nvidia Transformer Engine (TE) (Nvidia, 2022b) still relies on full-precision FP32 for collective communication, resulting in the same level of reduction for our FP8 solution.

When training GPT models with identical batch sizes, FP8 mixed-precision can lead to a reduction in memory footprint ranging from 27% to 42% when compared to BF16, as reported in Tab. 5. These reductions in memory consumption are attributed to the FP8 gradient and FP8 optimizer techniques we have introduced. Moreover, compared with TE (Nvidia, 2022b), our solution is also very competitive, obtaining 34.2%, 35.4%, and 44.8% additional memory reductions for different model sizes, *i.e.*, GPT-7B, 13B, and 175B. Although TE employs FP8 for compute, it still uses high-precision optimizer and gradients, which consumes much more memory than our solution. In addition, the saved memory in our method can be used to train larger batch size or longer sequence. For example, when employing 32 H100 GPUs with a memory capacity of 80G, our approach enables the training of models with a context of 4,096 tokens, accommodating up to 175 billion

Mixed-precision	System Performance		Model Performance	
	GPU Mem. (GB)	Throughput	AlpacaEval	MT-Bench
<b>BF16</b>	51.1	103	66.15	5.75
<b>FP8</b>	44.0( <small>-13.9%</small> )	131( <small>+27.2%</small> )	67.20	5.70

Table 3: A comparison between FP8 and BF16 for SFT. For system performance, we report results of GPU memory usage and training throughput. For model performance, we present the win-rate against Davinci-003 on AlpacaEval and GPT-4 judged scores on MT-Bench.

Mixed-precision	Memory Usage (MB)		Model Performance	
	Weights	Optimizer States	AlpacaEval	MT-Bench
<b>BF16</b>	15,082	15,116	72.05	6.16
<b>FP8</b>	10,292( <small>-31.8%</small> )	5,669( <small>-62.5%</small> )	72.42	6.04

Table 4: A comparison of FP8 and BF16 RLHF alignment. Memory usage is assessed with a focus on weights and optimizer states, while model performance is evaluated on AlpacaEval considering win-rate against Davinci-003, and MT-Bench using GPT-4 judged scores.

Model	TP	PP	DP	Micro BS	Mixed Precision	GPU Mem. (GB)	Throughput (#samples/s)	TFLOPS	MFU (%)	Weight-related Comm. Rate (%)	Volume (GB)
GPT-7B	1	1	32	2	BF16	69.6	158.1	442	44.7	9.2	37.2
				2	FP8 (TE)	77.2	219.4	613	31.0	9.5	37.2
				2	FP8 (Ours)	50.8 (-27%)	196.2 (+24%)	547	27.6	6.3	13.9 (-63%)
				4	FP8 (Ours)	69.0	215.6 (+36%)	602	30.4	8.3	13.9 (-63%)
GPT-13B	2	1	16	2	BF16	68.4	78.2	415	42.0	11.4	34.3
				2	FP8 (TE)	73.8	111.8	593	30.0	7.9	34.3
				2	FP8 (Ours)	47.7 (-30%)	99.3 (+27%)	498	25.9	5.8	12.4 (-64%)
				4	FP8 (Ours)	65.4	113.1 (+45%)	600	30.3	6.7	12.4 (-64%)
GPT-175B	8	4	4	1	BF16	63.4	22.5	388	39.2	8.8	23.4
				1	FP8 (TE)	66.7	31.4	541	27.3	4.3	23.4
				1	FP8 (Ours)	36.8 (+42%)	24.3 (+8%)	418	21.1	1.6	8.2 (-65%)
				4	FP8 (Ours)	52.3	36.8 (+64%)	634	32.0	8.9	8.2 (-65%)

Table 5: System-level performance on Nvidia H100 GPUs 80G. Here, TP, PP, and DP represent tensor, pipeline, and data parallelism respectively. BS indicates batch size, while MFU denotes model FLOPs utilization. Weight-related communication contains the all-gather operator on weights and the reduce-scatter operator on weight gradients.

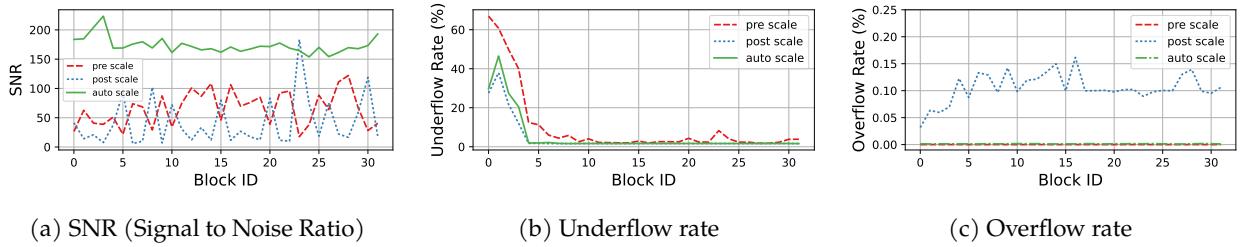


Figure 7: Comparing different strategies, *i.e.*, pre-scaling, post-scaling, and auto-scaling, for FP8 gradient all-reduce. We investigate SNR, underflow rate, and overflow rate across different Transformer blocks. The experiment is conducted using a GPT-7B model with a data parallelism factor of 128.

parameters. In contrast, TE can only accommodate models with a context of 2,048 tokens. This showcases the potential of integrating our FP8 mixed-precision training into existing LLMs, empowering them to train longer sequences with the same GPU resources.

Moreover, our FP8 mixed-precision scheme shows a superior training throughput compared to the prevalent BF16 scheme, achieving a notable speed-up of 64% when applied to GPT-175B model. The model FLOPS utilization (MFU) of FP8 mixed-precision training is 32.0% on H100 GPUs, being 17.2% superior to TE. These findings provide substantial evidence that our FP8 scheme effectively conserves memory, reduces communication costs during the training of large models, and ultimately enhances system utilization efficiency on the latest H100 GPU platform.

### 3.3 Ablation Study

We ablate various design choices of FP8 mixed-precision training strategy for LLMs and report the performance in Tab. 6 – 8 and Fig. 7 – 8. The ablation experiments are conducted on GPT models, whose architectures and training settings are elaborated in Tab. 1. Importantly, our ablation study yields several guidelines for the effective utilization of 8-bit datatype in LLM training, which can facilitate future research on low-bit model training.

*Communication.* We first analyze the limitations of the conventional pre-scaling and post-scaling methods when aggregating low-bit gradients during the all-reduce communication process. As shown in Fig. 7, we conduct a statistical analysis on SNR, underflow rate, and overflow rate of weight gradients across different Transformer blocks. It is observed that the pre-scaling method has relative larger underflow rate when quantifying gradients from 32-bit to 8-bit, while the post-scaling method has higher overflow rate. In contrast,

Low-bit Settings	Compute (GEMM)	Comm.	Master Weight	Optimizer States
FP32 #0	FP32	FP32	FP32	FP32+FP32
BF16 #1	BF16	FP32	FP32	FP32+FP32
FP8 #2	FP8	FP8	FP16	FP8+FP16
FP8 #3	FP8	FP8	FP8	FP8+FP16
FP8 #4	FP8	FP8	FP16	FP8+FP8

Table 6: Precision decoupling for the variables within the optimizer. Here, our focus is on ablating the master weight and optimizer states, as these components are precision sensitive. The optimizer states include both first-order and second-order gradient moments. Note that the FP16 master weight uses tensor scaling.

the proposed auto-scaling technique can diminish both the underflow ratio and the overflow ratio, while getting much better SNR, as shown in Fig. 7 (a). This demonstrates the effectiveness of auto-scaling method in reducing quantization errors when utilizing 8-bit datatype for gradient all-reduce.

*Optimizer.* We further ablate the impact of reduced precision for the variables in the AdamW optimizer. We set the BF16 mixed-precision optimizer as the baseline, since it has been widely used in existing LLM training frameworks (Micikevicius et al., 2017; Shoeybi et al., 2019; Nvidia, 2022b). Tab. 6 presents the settings of reduced precision for the variables, while Fig. 8 plots the corresponding training losses. We observe that: 1) FP8 master weight induces performance degradation (see the #2 vs. #3 lines in Fig. 8), while FP16 can maintain accuracy as FP32 (see #2 vs. #0 and #1) but requiring using tensor scaling. It reveals that the master weight is precision-sensitive. This can be attributed to the master weight’s role in updating weights, which tend to exhibit small magnitudes, necessitating high precision to maintain accuracy. 2) The second-order gradient moment is more precision-sensitive than the first-order one, because the square calculation is easy to cause underflow and leads to accuracy degradation. Utilizing FP8 for the second-order gradient moment can lead to divergent training loss (see the #4 dot in Fig. 8).

*Parallelism.* In our FP8 LLM training framework, we introduce FP8 low-bit convertors into sequence parallelism and tensor parallelism to reduce activation communication costs across GPUs. Here we conduct an analysis experiment to count the activation-related communication volume during GPT model training, and report the numbers in Tab. 7. It is observed that our FP8 parallel scheme results in a substantial reduction of 33% in activation-related communication costs compared to the original method utilizing BF16. Furthermore, in ZeRO distributed training, our method distributes each FP8 tensor along with its associated scaling factor as a whole, rather than partitioning the tensor into splits across GPUs. This

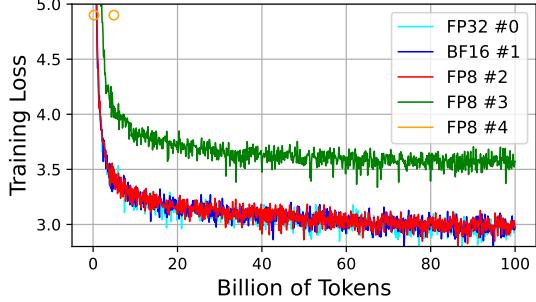


Figure 8: Training losses of GPT-125M models with the settings presented in Tab. 6. The loss curve for FP8 #4 has diverged.

Model	TP	PP	DP	Micro BS	Mixed Precision	Act-related Comm. Rate (%)	Volume (GB)
GPT-13B	2	1	16	2	BF16	11.1	4.7
					FP8 (Ours)	2.5	3.1
GPT-175B	8	4	4	1	BF16	14.8	5.9
					FP8 (Ours)	4.5	3.9

Table 7: Activation-related communication volume reduction in sequence and tensor parallelism, including the all-gather operator on activation and the reduce-scatter on activation gradients.

Model	TP	PP	DP	Micro BS	Mixed Precision	GPU Memory Min	Max
GPT-7B	1	1	32	2	BF16	69.02	69.59
					FP8 (TE)	76.93	77.23
					FP8 (Ours)	49.31	50.77
GPT-13B	2	1	16	2	BF16	67.63	68.35
					FP8 (TE)	72.85	73.81
					FP8 (Ours)	46.78	47.66
GPT-175B	8	4	4	1	BF16	63.07	63.42
					FP8 (TE)	66.39	66.72
					FP8 (Ours)	36.39	36.81

Table 8: Comparing ZeRO distribution methods in terms of memory load across GPUs. Here “Min” and “Max” denote the minimum and maximum memory utilization observed across 32 GPUs. Our FP8 ZeRO method uses less memory while achieving memory-aware load balancing.

activation-related communication volume during GPT model training, and report the numbers in Tab. 7. It is observed that our FP8 parallel scheme results in a substantial reduction of 33% in activation-related communication costs compared to the original method utilizing BF16. Furthermore, in ZeRO distributed training, our method distributes each FP8 tensor along with its associated scaling factor as a whole, rather than partitioning the tensor into splits across GPUs. This

strategy not only results in more GPU memory savings but also maintains a balanced memory load across GPUs, as demonstrated in Tab. 8.

## 4 Related Work

**Mixed-precision Training.** Efficient training through reduced mixed-precision has been widely used in modern deep learning to save computing costs. While some works have taken bit-reduction to the extreme, *i.e.* 1-bit binary networks (Hubara et al., 2016; Rastegari et al., 2016), they have not been successful in maintaining model accuracy (Micikevicius et al., 2022). The most practical scheme now is the FP16 half-precision method (Micikevicius et al., 2017), which can maintain accuracy while improving training efficiency. The computations during forward pass and back propagation use FP16 while the master weights use FP32. Since FP16 has a narrower dynamic range, FP16 mixed-precision entails loss scaling (Micikevicius et al., 2017) to prevent loss of accuracy. Fortunately, the need for loss scaling can be avoided by using BF16 datatype, because BF16 maintains the same dynamic range as the full-precision FP32. This results in that large model training now prefers to use BF16 mixed-precision scheme, which is more stable during training (Smith et al., 2022; Scao et al., 2022; Zeng et al., 2022).

FP8 is a natural progression from 16-bit data formats to further reducing computing cost. Early pioneering efforts in FP8 low-bit model training (Wang et al., 2018; Sun et al., 2019; Dettmers et al., 2021) have largely remained at the simulation stage. Consequently, there exists a notable gap between the projected capabilities of these approaches and their actual performance on hardware (Micikevicius et al., 2022). With the advent of Nvidia Hopper GPU architecture (Nvidia, 2022a), FP8 is emerging as a viable and practical data type for the next-generation low-precision training, as discussed in (Micikevicius et al., 2022). At present, the Nvidia Transformer Engine (TE) (Nvidia, 2022b) serves as the primary framework for FP8 mixed-precision training. However, its support for FP8 usage remains somewhat constrained. TE’s current implementation restricts FP8 usage solely to weight computation, retaining the storage of model weights and gradient calculations with 16-bit data types. Consequently, the end-to-end speed-up, memory and communication cost savings are limited. In contrast, our work infiltrates FP8 gradient, optimizer, and distributed training into the whole progress of model training, fully unveiling the capabilities of FP8.

**Large Language Models.** Recent years have witnessed a substantial evolution in the field of LLMs. Autoregressive language modeling – predicting the future of a text sequence from its past – provides a simple yet powerful objective that admits formulation of numerous tasks. While there exist alternative methodologies, such as masked language modeling (Devlin et al., 2019) and permutation language modeling (Yang et al., 2019), the autoregressive method now is more promising because of its strong performance. Following the scaling laws (Brown et al., 2020) and the refined laws (Hoffmann et al., 2022), various LLMs are have been proposed, including dense models: GPT-3 (Brown et al., 2020), Jurassic-1 (Lieber et al., 2021), Gopher (Rae et al., 2021), Chinchilla (Hoffmann et al., 2022), Bloom (Scao et al., 2022), OPT (Zhang et al., 2022) Megatron-Turing NLG (Smith et al., 2022), PaLM (Chowdhery et al., 2022), LaMDA (Thoppilan et al., 2022), LLaMA (Touvron et al., 2023), and sparse models: GLaM (Du et al., 2022), and Switch transformers (Fedus et al., 2022). Each of them has demonstrated remarkably competitive few-shot performance across a wide range of tasks at the time of their respective releases. Nonetheless, these models still encounter challenges, such as overwhelming computational requirements and the need for acquiring more high-quality training data. In this work, we delve into the utilization of low-precision techniques to mitigate the training costs, which is a crucial step for the continued expansion of language models.

Low-precision training has been widely used in LLM training to reduce compute cost. OPT (Zhang et al., 2022) and GLM (Zeng et al., 2022) utilize FP16 for forwards and backwards and FP32 for optimizer states and master weights, to reduce the GPU memory usage and improve training efficiency. Bloom (Scao et al., 2022) find that FP16 can cause numerical instabilities and irreversible divergences, especially when training models larger than 100B parameters, because FP16’s dynamic range is limited. Consequently, Bloom and other LLMs, such as Gopher (Rae et al., 2021) and Chinchilla (Hoffmann et al., 2022), adopt BF16 mixed-precision, because BF16 has a wide dynamic range that is the same as FP32. LLM training and tuning with 8-bit low-precision were not well-explored in previous works, because the hardware support for FP8 is not available before the release of Nvidia Hopper infrastructure. This work presents the first exploration of FP8 pre-training

and fine-tuning for LLMs, while proposing an extremely-optimized FP8 mixed-precision scheme. We hope this work could facilitate future research in FP8 and, potentially, extend to exploring even lower precision training, such as 4-bit and 1-bit.

## 5 Conclusion

In this work, we explore 8-bit training for LLMs. We introduce a new FP8 mixed-precision training framework, which incorporates 8-bit collective communication, optimizer, and distributed parallel training in an incremental manner. To our best knowledge, this is the first work infiltrating FP8 compute, storage and communication into the whole progress of large language model training. Extensive experiments demonstrate the proposed method effectively diminishes communication overhead and curtails memory utilization in the context of GPT model training at various scales. In future work, we plan to scale up the size and training steps of the FP8 GPT models and further train them with our 8-bit mixed-precision scheme. Moreover, we will also use the proposed FP8 scheme to train multi-modal large models, and explore low-bit deployment of LLMs on various edge devices, such as smart phones.

## Contribution and Acknowledgement

This project was initially proposed by Han Hu and Peng Cheng, who are the directional lead. Shuguang Liu served as the product lead throughout the project.

The contributions for all the co-authors are detailed as follows:

**FP8 Framework:** Kan Wu, Houwen Peng, Ze Liu, Peng Cheng, Han Hu

**System:** Yifan Xiong, Ziyue Yang, Yuxiang Yang, Guoshuai Zhao, Peng Cheng

**Hardware Infrastructure:** Guoshuai Zhao, Yuxiang Yang, Yifan Xiong, Peng Cheng, Shuguang Liu, Joe Chau

**Data:** Ruihang Li, Miaosen Zhang, Jia Ning, Chen Li, Ruizhe Wang, Houwen Peng, Han Hu

**Pre-training:** Yixuan Wei, Kan Wu, Ze Liu, Miaosen Zhang, Zheng Zhang, Houwen Peng, Han Hu

**Alignment (SFT, RS, and RLHF):** Bolin Ni, Jingcheng Hu, Yixuan Wei, Houwen Peng, Han Hu

**Evaluation:** Yixuan Wei, Bolin Ni, Jingcheng Hu

**Product Engineering:** Yuxiang Yang, Kan Wu, Yifan Xiong, Ziyue Yang, Guoshuai Zhao, Peng Cheng

We thank Eric Chung, Bita Darvish Rouhani, Yu Pei, Hyunseung Harry Yoo, Zhenghong Zhou, Gongrui Zhang, and Zhirong Wu for helpful discussions.

We thank Baining Guo and Lidong Zhou for their guidance and support for this project.

## References

- Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.
- Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.
- Microsoft Bing. Bing webmaster tools. 2022. URL <https://www.bing.com/webmasters/>.
- Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439, 2020.
- Sidney Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. In *Proceedings of BigScience Episode# 5—Workshop on Challenges & Perspectives in Creating Large Language Models*, pages 95–136, 2022.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.
- Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek B Rao, Parker Barnes, Yi Tay, Noam M. Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Benton C. Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier García, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Oliveira Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Díaz, Orhan Firat, Michele Catasta, Jason Wei, Kathleen S. Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311, 2022.
- Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. BoolQ: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2924–2936, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1300. URL <https://aclanthology.org/N19-1300>.
- Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv:1803.05457v1*, 2018.
- Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.
- Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. In *International Conference on Learning Representations*, 2021.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*,

pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL <https://www.aclweb.org/anthology/N19-1423>.

Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, pages 5547–5569. PMLR, 2022.

Yann Dubois, Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. Alpacafarm: A simulation framework for methods that learn from human feedback. *arXiv preprint arXiv:2305.14387*, 2023.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research*, 23(1):5232–5270, 2022.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. <http://Skylion007.github.io/OpenWebTextCorpus>, 2019.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv:2203.15556*, 2022.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in neural information processing systems*, 29, 2016.

HuggingFace. wikipedia - datasets at hugging face. 2022. URL <https://huggingface.co/datasets/wikipedia>.

Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefer. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021.

Armand Joulin, Édouard Grave, Piotr Bojanowski, and Tomáš Mikolov. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 427–431, 2017.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations*, San Diego, CA, 2015. URL <http://arxiv.org/abs/1412.6980>.

Denis Kocetkov, Raymond Li, LI Jia, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, et al. The stack: 3 tb of permissively licensed source code. *Transactions on Machine Learning Research*, 2022.

Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi-Rui Tam, Keith Stevens, Abdullah Barhoum, Nguyen Minh Duc, Oliver Stanley, Richárd Nagyfi, et al. Openassistant conversations—democratizing large language model alignment. *arXiv preprint arXiv:2304.07327*, 2023.

Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. Deduplicating training data makes language models better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8424–8445, 2022.

Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 766–775, 2023a.

Xuechen Li, Tianyi Zhang, Yann Dubois, Rohan Taori, Ishaan Gulrajani, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Alpacaeval: An automatic evaluator of instruction-following models. [https://github.com/tatsu-lab/alpaca\\_eval](https://github.com/tatsu-lab/alpaca_eval), 2023b.

Opher Lieber, Or Sharir, Barak Lenz, and Yoav Shoham. Jurassic-1: Technical details and evaluation. *White Paper. AI21 Labs*, 1, 2021.

- Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.
- Ze Liu, Han Hu, Yutong Lin, Zhuliang Yao, Zhenda Xie, Yixuan Wei, Jia Ning, Yue Cao, Zheng Zhang, Li Dong, et al. Swin transformer v2: Scaling up capacity and resolution. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12009–12019, 2022.
- Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2018.
- Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.
- Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al. Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.
- Microsoft. Azure high-performance computing. 2023. URL <https://azure.microsoft.com/en-us/solutions/high-performance-computing>.
- Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2381–2391, 2018.
- Nvidia. Apex. 2018. URL <https://nvidia.github.io/apex>.
- Nvidia. The nvidia collective communications library. 2020. URL <https://developer.nvidia.com/nccl>.
- Nvidia. Nvidia h100 tensor core gpu architecture. 2022a. URL <https://resources.nvidia.com/en-us-tensor-core>.
- Nvidia. Nvidia transformer engine. 2022b. URL <https://docs.nvidia.com/deeplearning/transformer-engine/index.html>.
- Nvidia. Using fp8 with transformer engine. 2022c. URL [https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8\\_primer.html](https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html).
- OpenAI. Model index for researchers. 2022. URL <https://platform.openai.com/docs/model-index-for-researchers>.
- OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc-Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1525–1534, 2016.
- Shawn Presser. Books3. <https://twitter.com/theshawwn/status/1320282149329784833>, 2020.
- Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.
- Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, Chloe Hillier, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*, 2019.
- Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.
- Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

- Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.
- Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International Conference on Machine Learning*, pages 8821–8831. PMLR, 2021.
- Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.
- Redpajama. Redpajama-data: an open source recipe to reproduce llama training dataset. 2023. URL <https://github.com/togethercomputer/RedPajama-Data>.
- Melissa Roemmele, Cosmin Adrian Bejan, and Andrew S. Gordon. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *AAAI Spring Symposium*, 2011.
- Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.
- David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *International Conference on Learning Representations*, 2018.
- Teven Le Scao, 388 Authors, and Thomas Wolf. BLOOM: A 176B-parameter open-access multilingual language model. *ArXiv*, abs/2211.05100, 2022.
- John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- ShareGPT. Openchat: Advancing open-source language models with imperfect data. 2023. URL <https://sharegpt.com/>.
- Mohammad Shoeybi, Mostafa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- Shaden Smith, Mostafa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.
- Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.
- Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Viji Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. *Advances in neural information processing systems*, 32, 2019.
- Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi Viji Srinivasan, and Kailash Gopalakrishnan. Ultra-low precision 4-bit training of deep neural networks. *Advances in Neural Information Processing Systems*, 33:1796–1807, 2020.
- Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.
- Jörg Tiedemann. Finding alternative translations in a large corpus of movie subtitle. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 3518–3522, 2016.
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- Trieu H Trinh and Quoc V Le. A simple method for commonsense reasoning. *arXiv preprint arXiv:1806.02847*, 2018.

- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008. Curran Associates, Inc., 2017. URL <http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf>.
- VicunaTeam. Vicuna: An open-source chatbot impressing gpt-4 with 90quality. 2023. URL <https://lmsys.org/blog/2023-03-30-vicuna/>.
- Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. *Advances in neural information processing systems*, 31, 2018.
- Guillaume Wenzek, Marie-Anne Lachaux, Alexis Conneau, Vishrav Chaudhary, Francisco Guzmán, Armand Joulin, and Edouard Grave. Ccnet: Extracting high quality monolingual datasets from web crawl data. *arXiv preprint arXiv:1911.00359*, 2019.
- Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. XLNet: Generalized autoregressive pretraining for language understanding. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL <https://proceedings.neurips.cc/paper/2019/file/dc6a7e655d7e5840e66733e9ee67cc69-Paper.pdf>.
- Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4791–4800, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1472. URL <https://aclanthology.org/P19-1472>.
- Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. Glm-130b: An open bilingual pre-trained model. In *The Eleventh International Conference on Learning Representations*, 2022.
- Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuhui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *arXiv preprint arXiv:2306.05685*, 2023.
- Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.

## A Appendix

### A.1 FP8 Data Formats

In September 2022, NVIDIA, ARM, and Intel published FP8 specification for standardization as an interchange format for AI (Micikevicius et al., 2022). The industry has moved from 32-bit precision to 16-bit, and now even 8-bit precision for AI model training. This development reflects a broader industry trend that has transitioned from high-precision to low-precision training. Notably, the proposed FP8 specification introduces two distinct data types, *E5M2* and *E4M3*, which offer a trade-off between a larger range and higher precision of stored values (Nvidia, 2022c).

- *E4M3* consists of 1 sign bit, 4 exponent bits and 3 bits of mantissa. It can store values up to +/-448 and NaN.
- *E5M2* consists of 1 sign bit, 5 exponent bits and 2 bits of mantissa. It can store values up to +/-57344, +/- inf and NaN.

The FP8 format (Micikevicius et al., 2022) roughly follows the IEEE 754 standard. Compared to higher precision data formats such as FP16 and FP32, FP8 suffers from two kinds of representation degradation:

- *Lower representation range*. The representation range in a data format specifies the range between the maximum and minimum values that the format can accurately represent. There are two modes, a normal mode, which defines a regular range with relatively constant precision, and a subnormal mode, which extends the range to represent smaller values with lower precision. The normal range primarily depends on the number of exponent (*E*) bits, with more *E* bits resulting in a larger normal range. On the other hand, the subnormal range is primarily influenced by the number of mantissa (*M*) bits, where an increase in *M* bits leads to a larger subnormal range. As illustrated in Tab. 9, the representation range of FP8 is notably narrower compared to that of FP16 and FP32, especially in the case of the *S1E4M3* sub-format (*S* denotes the sign bit). This discrepancy represents the primary challenge when employing FP8 for training large models.
- *Lower representation precision*. The limited number of mantissa (*M* bits) leads to quantization representation errors. Due to the considerably fewer *M* bits in FP8, the representation precision of FP8 is substantially lower than that of FP16, as depicted in Tab. 9. This challenge stands as another significant hurdle when considering the use of FP8 for training large models.

FP8 consists of two sub-formats: *S1E4M3* and *S1E5M2*. The former offers a narrower representation range but higher precision, while the latter provides a larger range but lower precision. These two sub-formats give users the flexibility to strike a balance between their requirements for range and precision in model training.

Table 9: Representation range and error for different data formats

Data format	Representation Range			Maximum Relative Error	
	Max normal	Min normal	Min subnormal	Min - Max (normal)	Min ~ Max (subnormal)
FP32 (S1E8M23)	$3.40 \times 10^{38}$	$1.18 \times 10^{-38}$	$1.40 \times 10^{-45}$	$1.19 \times 10^{-7} \sim 5.96 \times 10^{-8}$	$5.00 \times 10^{-1} \sim 1.19 \times 10^{-7}$
FP16 (S1E5M10)	65,504	$6.10 \times 10^{-5}$	$5.96 \times 10^{-8}$	$9.76 \times 10^{-4} \sim 4.89 \times 10^{-4}$	$5.00 \times 10^{-1} \sim 9.78 \times 10^{-4}$
BF16 (S1E8M7)	$3.39 \times 10^{38}$	$1.18 \times 10^{-38}$	$9.18 \times 10^{-41}$	$7.75 \times 10^{-3} \sim 3.94 \times 10^{-3}$	$5.00 \times 10^{-1} \sim 7.94 \times 10^{-3}$
FP8 (S1E4M3)	448	$1.56 \times 10^{-2}$	$1.95 \times 10^{-3}$	$1.11 \times 10^{-1} \sim 7.69 \times 10^{-2}$	$5.00 \times 10^{-1} \sim 1.67 \times 10^{-1}$
FP8 (S1E5M2)	57,344	$6.10 \times 10^{-5}$	$1.53 \times 10^{-5}$	$2.00 \times 10^{-1} \sim 1.67 \times 10^{-1}$	$5.00 \times 10^{-1} \sim 5.00 \times 10^{-1}$

## A.2 FP8 Tensor Scaling

We now discuss the underlying mechanisms for how large model training with FP8 overcomes the challenges associated with representation range and precision degradation. The key technique behind is tensor scaling, which scales the tensor values that originally locate out the representation range of a data format to its comfort zone, as visualized in Fig. 9. The pioneer scaling techniques (Micikevicius et al., 2017; Nvidia, 2018) apply a global scaling factor to the loss, such that gradients of all layers are scaled by a single adaptive factor. The utilization of the global loss scaling technique, in conjunction with various other training strategies, has facilitated the widespread adoption of FP16 mixed-precision training on V100 and A100 GPUs. Remarkably, this approach has resulted in minimal to no degradation in accuracy, particularly for small to medium-sized models (Micikevicius et al., 2017). Nonetheless, when dealing with super-large models or complex tasks, such as in the training of models like DALL-E (Ramesh et al., 2021), the global loss scaling technique still encounters significant underflow issues. As a consequence, block-wise (Ramesh et al., 2021) and layer-wise (Sun et al., 2020) gradient scaling are proposed.

While the global scaling technique enables almost no accuracy drop for FP16 training (with a range of [5.96E-8, 6.55E+4]), the fine-grained per-tensor scaling will enable stable model training using even shallower range by FP8 (with a range of [1.95E-3, 448] for E4M3 and a range of [1.53E-5, 5.73E+4] for E5M2). Fig. 9 shows that the representation range of FP8 has been large enough to deal with general model training. In the per-tensor scaling technique, various strategies are available for choosing the suitable scaling factor for a given FP8 tensor. Two common approaches are “just-in-time scaling” and “delayed scaling” (Nvidia, 2022c).

- *Just-in-time scaling*. This strategy involves determining the scaling factor based on the maximum absolute value (amax) of the tensor being generated. However, in practical applications, this approach is often infeasible because it necessitates multiple passes through the data. Specifically, the operator first produces and writes out the output in higher precision, then calculates the maximum absolute value of the output, and finally applies this scaling factor to all values to obtain the final FP8 output. This process introduces a significant amount of overhead, which can substantially reduce the benefits of using FP8.
- *Delayed scaling*. This strategy involves selecting the scaling factor based on the maximum absolute values observed in a certain number of preceding iterations. This approach allows for the full performance benefits of FP8 computation but necessitates the storage of a history of maximum values as additional parameters of the FP8 operators.

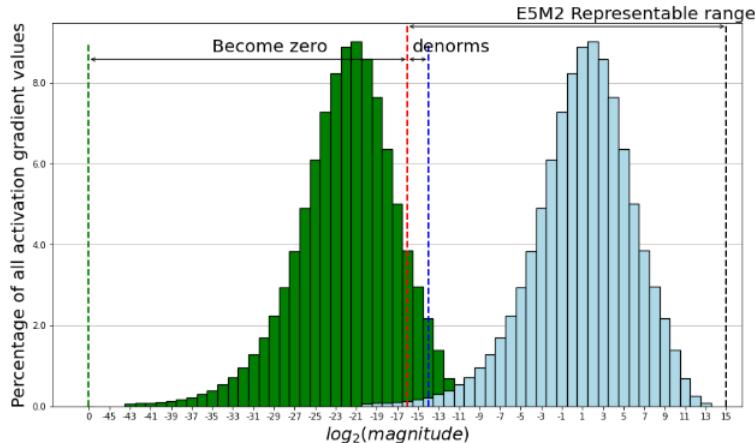


Figure 9: Scaling gradients to fall within the representation range of the FP8 datatype.

### A.3 Pre-training Data

Tab. 10 presents an overview of our collected data sources along with the corresponding sampling weights employed in pre-training. The arXiv and StackExchange subsets are collected from Redpajama (Redpajama, 2023), while BookCorpus2 (Zhu et al., 2015), Books3 (Presser, 2020), DM-Math (Saxton et al., 2018), Gutenberg (Rae et al., 2019), HackerNews<sup>6</sup>, NIH ExPorter<sup>7</sup>, OpenSubtitles (Tiedemann, 2016), and USPTO<sup>8</sup> subsets are extracted from The Pile (Gao et al., 2020). The Wikipedia data is downloaded from HuggingFace (HuggingFace, 2022). We use the 20220301 dump, including 24 languages: bg, ca, cs, da, de, en, es, fr, hi, hr, hu, it, jp, ko, nl, pl, pt, ro, ru, sl, sr, sv, uk, zh.

We pre-process 11 CommonCrawl snapshots, ranging from 2018 to 2023, with the CCNet pipeline (Wenzek et al., 2019). This process involves data deduplication at the line level, followed by language identification utilizing a fastText linear classifier (Joulin et al., 2017) to eliminate non-English pages. A filtering mechanism based on an n-gram language model is employed to exclude low-quality content. In addition, we train a linear classifier (Redpajama, 2023) to distinguish documents similar to Wikipedia pages from randomly sampled CommonCrawl documents. Documents not classified as resembling Wikipedia are excluded. Finally, we perform fuzzy deduplication (Lee et al., 2022) across all the processed snapshots from CommonCrawl.

We collect Python code data from Github using a repository list provided by Bing indexing (Bing, 2022). The cleaning of the code data includes three steps. First, we remove control characters, except for \t and \n. Next, we remove copyright comments in the code. An alphanumeric rate filter is then applied, removing lines with a rate below 0.5 if they are comments, and discarding the entire file if its overall alphanumeric rate is less than 0.98. Files with less than 5 lines or a maximum line length exceeding 1,000 characters are also discarded. Also, files with an average line length of more than 100 characters are discarded. Lastly, a pattern search is conducted to identify key Python keywords (*e.g.*, import, from, def, class, if, for, try, etc.) within the code. Files containing less than 3 instances of these keywords are eliminated. This comprehensive process ensures that the remaining Python code data is of high quality and suitable for use in academic research. We additionally add Python code from Stack (Kocetkov et al., 2022), and perform fuzzy deduplication within all the collected Python code.

---

<sup>6</sup><https://news.ycombinator.com>

<sup>7</sup><https://exporter.nih.gov>

<sup>8</sup><https://bulkdata.uspto.gov>

Dataset	Sampling prop.	Epochs	Training Tokens (Billion)
<b>Web Crawls</b>			
CommonCrawl	51.71%	0.16	51.71
C4	25.56%	0.16	25.56
OpenWebText	2.73%	0.16	2.73
<b>Technical &amp; Science content</b>			
ArXiv	1.54%	0.05	1.54
StackExchange	1.42%	0.08	1.42
DM-Math	0.39%	0.05	0.39
USPTO	0.52%	0.05	0.52
NIH ExPorter	0.04%	0.05	0.04
<b>Programming Languages</b>			
Python	4.50%	0.11	4.50
<b>Other Curated Sources</b>			
Wikipedia	4.50%	0.16	4.50
Books	4.50%	0.09	4.50
News	2.00%	0.11	2.00
Dialogue	2.00%	0.27	2.00
<b>Total</b>			100.00

Table 10: Pre-training data. For each subset we list the sampling weight, number of epochs, and training tokens. Books data includes BookCorpus2 ([Zhu et al., 2015](#)), Books3 ([Presser, 2020](#)), and Gutenberg ([Rae et al., 2019](#)). Dialogue data includes HackerNews and OpenSubtitles ([Tiedemann, 2016](#)). For experiments with a training token count of less than 100 billion, we employ the same sampling proportion as shown in this table.

---

# FP8 FORMATS FOR DEEP LEARNING

---

**Paulius Micikevicius, Dusan Stosic, Patrick Judd, John Kamalu, Stuart Oberman, Mohammad Shoeybi,  
Michael Siu, Hao Wu**

NVIDIA

{pauliusm, dstosic, pjudd, jkamalu, soberman, mshoeybi, msiu, skyw}@nvidia.com

**Neil Burgess, Sangwon Ha, Richard Grisenthwaite**

Arm

{neil.burgess, sangwon.ha, richard.grisenthwaite}@arm.com

**Naveen Mellemudi, Marius Cornea, Alexander Heinecke, Pradeep Dubey**

Intel

{naveen.k.mellemudi, marius.cornea, alexander.heinecke, pradeep.dubey}@intel.com

## ABSTRACT

FP8 is a natural progression for accelerating deep learning training inference beyond the 16-bit formats common in modern processors. In this paper we propose an 8-bit floating point (FP8) binary interchange format consisting of two encodings - E4M3 (4-bit exponent and 3-bit mantissa) and E5M2 (5-bit exponent and 2-bit mantissa). While E5M2 follows IEEE 754 conventions for representation of special values, E4M3's dynamic range is extended by not representing infinities and having only one mantissa bit-pattern for NaNs. We demonstrate the efficacy of the FP8 format on a variety of image and language tasks, effectively matching the result quality achieved by 16-bit training sessions. Our study covers the main modern neural network architectures - CNNs, RNNs, and Transformer-based models, leaving all the hyperparameters unchanged from the 16-bit baseline training sessions. Our training experiments include large, up to 175B parameter, language models. We also examine FP8 post-training-quantization of language models trained using 16-bit formats that resisted fixed point int8 quantization.

## 1 Introduction

Continued improvement in state of the art deep learning (DL) results has required continued increase in neural network model sizes and compute resources needed to train them. For example, large natural language models such as GPT-3 [2], Turing-Megatron [18], PaLM [4], and OPT [25] take weeks to train on thousands of processors. Reduced precision representation of numbers has been the cornerstone for deep learning training and inference acceleration. Common floating point types for training include IEEE single precision, TF32 mode for single precision [19], IEEE half precision [14], and bfloat16 [9]. While some research publications have taken bit-reduction to the extreme, i.e. 1-bit binary networks [5, 7, 26, 24, 17], they have not been successful in maintaining result quality needed for many practical applications. For inference fixed-point int8 representation is a popular option. In some cases even int8 inference can encounter challenges in maintaining the accuracy required for application deployment [1]. Additional number representations, such as log-format [15, 11], posit, and log with posit exponent values [8] have been proposed in literature but have not been adopted in practice because the demonstrated benefits have not been sufficient to justify new math pipeline hardware designs.

FP8 is a natural progression from 16-bit floating point types, reducing the compute requirements of neural network training. Furthermore, due to its non-linear sampling of the real numbers, FP8 can also have advantages for inference when compared to int8. Wang et al. [22] proposed using 5-bit exponent format for training neural networks, confirming their methodology on the convolutional neural networks (CNNs) for image classification on CIFAR-10 and ILSVRC12

datasets. Mellemudi et al. [12] study the 5-bit exponent format for training on the larger CNNs as well as language translation networks based on recurrent and transformer blocks. Both papers investigate 16-bit weight updates as well as stochastic rounding. Use of two FP8 formats, 4- and 5-bit exponent fields, for training is introduced in [20], studying a wider range of CNNs as well as speech and language translation models. [20] also investigates FP8 inference of networks trained in higher precision and introduces returning of batch normalization statistics to improve result accuracy. Noune et al [16] propose a modified FP8 representation that dedicates a single encoding to special values in order to increase the represented dynamic range and present an extensive study of exponent bias effect on result quality. 8-bit inference with various formats, including FP8, with networks trained in higher precision is the focus of [10].

In this paper we describe an 8-bit binary format for floating point representation, using two encodings for FP8. Basic principles of using FP8 for deep learning are summarized in Section 2. In Section 3 we describe the bit encodings and reasoning behind them. Empirical evaluation of training and inference with a variety of tasks and models is presented in Section 4. We show that FP8 training matches FP16 or bfloat16 training results for a variety of tasks and neural network model architectures and sizes, without changing any model or optimizer hyperparameters. Our study includes the training of very large language models, up to 175B parameters. It is important to consider a wide range of model sizes since it has been shown that models of different sizes may have different numerical behaviors (for example, the different behavior of Resnet-18 and Resnet-50 observed in [12]).

## 2 Aspects of FP8 Usage in Deep Learning

Some aspects of FP8 usage affect the choices for binary interchange format. For example, the dynamic ranges required by various networks dictate the need for the two formats as well as the preference for scaling factor handling in software rather than via exponent bias. Other aspects, such as type conversion specifics are orthogonal to the binary format. Both aspects are briefly reviewed in this section.

It is expected that mathematical operations on FP8 inputs will produce outputs in higher precision, optionally converting the results to FP8 prior to writing them to memory. This is common practice today for the 16-bit floating point formats (FP16 and bfloat16) found on CPUs, GPUs, and TPUs [3, 14]. For example, matrix-multiplication or dot-product instructions produce single-precision outputs but less arithmetically-intensive operations are typically performed after casting the 16-bit inputs to single precision. Thus, FP8 tensors will be generated by converting to FP8 from wider types, such as single precision floating point.

Higher precision values need to be multiplied with a scaling factor prior to their casting to FP8 in order to move them into a range that better overlaps with the representable range of a corresponding FP8 format. This is very similar to the purpose loss-scaling serves in mixed-precision training with FP16, where gradients are moved into FP16-representable range [14],[13](slides 13-16). However, some networks require per-tensor scaling factors as FP8 dynamic range is not sufficient to cover the union of all tensors' important values (see Section 3.2). Details of the heuristics to select the scaling factors are beyond the scope of this paper, but the general idea is to choose a scaling factor such that the maximum magnitude in the tensor becomes close to the maximum representable magnitude in the corresponding format. Values that overflow are then saturated to the maximum representable value. Weight update skipping (and reduction of the scaling factor) on overflows, as used by FP16 automatic mixed precision training [13], is not a good choice for FP8 as overflows are much more likely due to the narrower dynamic range, resulting in too many skipped updates. Values in higher precision get unscaled by multiplying with the inverse of the scaling factor, either after conversion from FP8 or after arithmetic instructions for a linear operation have produced a higher-precision output. In both cases only a minimal amount of additional arithmetic is required. For matrix multiplications, unscaling is applied once per dot-product, thus amortized by many multiply-accumulates with FP8 inputs. Less arithmetic intensive operations (such as nonlinearities, normalizations, or weight updates by optimizers) are typically memory-bandwidth limited and not sensitive to one additional arithmetic instruction per value.

While the mechanics of type conversion are orthogonal to the binary format, we briefly touch on some aspects for completeness of DL uses. Converting a special value from a wider precision to FP8 results in the corresponding special value in FP8. For conversions to E4M3 this means that both infinities and NaNs in the wider type (for example, single precision) turn into NaNs in FP8. This handling of special values is needed when using mixed precision training that involves both FP8 and FP16 types, since automatic mixed precision [13] run-time adjustment of loss-scaling relies on causing and detecting overflows. In addition, non-saturating mode of conversion can be provided for usecases that may require a strict handling of overflows. Rounding mode (round to nearest even, stochastic, etc.) choice is orthogonal to the interchange format is left up to the implementation, software and possibly hardware, for maximum flexibility.

### 3 FP8 Binary Interchange Format

FP8 consists of two encodings - E4M3 and E5M2, where the name explicitly states the number of exponent (E) and mantissa (M) bits. We use the common term "mantissa" as a synonym for IEEE 754 standard's trailing significand field (i.e. bits not including the implied leading 1 bit for normal floating point numbers). The recommended use of FP8 encodings is E4M3 for weight and activation tensors, and E5M2 for gradient tensors. While some networks can train with just the E4M3 or the E5M2 type, there are networks that require both types (or must maintain many fewer tensors in FP8). This is consistent with findings in [20, 16], where inference and forward pass of training use a variant of E4M3, gradients in the backward pass of training use a variant of E5M2.

FP8 encoding details are specified in Table 1. We use the *S.E.M* notation to describe binary encodings in the table, where *S* is the sign bit, *E* is the exponent field (either 4 or 5 bits containing biased exponent), *M* is either a 3- or a 2-bit mantissa. Values with a 2 in the subscript are binary, otherwise they are decimal.

Table 1: Details of FP8 Binary Formats

	E4M3	E5M2
Exponent bias	7	15
Infinities	N/A	$S.11111.00_2$
NaN	$S.11111.111_2$	$S.11111.\{01, 10, 11\}_2$
Zeros	$S.0000.000_2$	$S.00000.00_2$
Max normal	$S.1111.110_2 = 1.75 * 2^8 = 448$	$S.11110.11_2 = 1.75 * 2^{15} = 57,344$
Min normal	$S.0001.000_2 = 2^{-6}$	$S.00001.00_2 = 2^{-14}$
Max subnorm	$S.0000.111_2 = 0.875 * 2^{-6}$	$S.00000.11_2 = 0.75 * 2^{-14}$
Min subnorm	$S.0000.001_2 = 2^{-9}$	$S.00000.001_2 = 2^{-16}$

Design of these FP8 format followed the principle of staying consistent with IEEE-754 conventions, deviating only if a significant benefit is expected for DL application accuracy. Consequently, the E5M2 format follows the IEEE 754 conventions for exponent and special values and can be viewed as IEEE half precision with fewer mantissa bits (similar to how bfloat16 and TF32 can be viewed as IEEE single precision with fewer bits). This allows for straightforward conversion between E5M2 and IEEE FP16 formats. By contrast, the dynamic range of E4M3 is extended by reclaiming most of the bit patterns used for special values because in this case the greater range achieved is much more useful than supporting multiple encodings for the special values.

#### 3.1 Special value representations

We extend the narrow dynamic range of the E4M3 format by representing fewer special values, adopting their bit patterns for normal values. Infinities are not represented (see Section 2 for overflow handling details) and we retain only one mantissa bit-pattern for NaNs. This modification extends the dynamic range by one extra power of 2, from 17 to 18 binades. We gain the representation of seven more magnitudes (256, 288, 320, 352, 384, 416, 448), corresponding to the biased exponent value  $1111_2$ . The maximum representable magnitude without this modification would be 240. For consistency with IEEE 754 conventions we retain positive and negative representations for zero and NaN. While we could gain one additional representable magnitude, 480, by having just one encoding for zero and one for NaN, this would require breaking the symmetry of positive and negative representations inherent in the IEEE 754 formats, complicating or invalidating algorithm implementations that rely on this property. For example, IEEE floating point formats allow comparison and sorting of floating point values using integer operations. The benefit of increasing the maximum value from 448 to 480 for DL is not significant to warrant deviating from IEEE convention and losing software implementations that rely on it.

As mentioned earlier, E5M2 represents all the special values (infinities, NaNs, and zeros) consistently with IEEE conventions. Our extensive empirical studies (Section 4) indicate that 5 bits of exponent provide sufficient per tensor dynamic range (32 binades, including the subnormal values) for DL. Furthermore, the benefit of having fewer representations of special values would be much smaller for E5M2 than it was for E4M3 - only 3 additional magnitude values would be added due to the smaller mantissa, one additional binade is much less impactful when E5M2 already provides 32 (compared to E4M3's 17 without the adjustment).

### 3.2 Exponent bias

Both E4M3 and E5M2 retain IEEE-like exponent biases: 7 and 15 for E4M3 and E5M2, respectively. Exponent bias controls the placement of representable range on the real number line. The same effect is achieved when maintaining a scale factor per tensor. Our experiments indicate that there are neural networks that cannot use the same exponent bias for all tensors of a given type, requiring a per-tensor adjustment. One such example is discussed in Section 4.3. Consequently, we chose to not deviate from the IEEE convention for exponent bias. Leaving the per-tensor scaling to software implementation enables more flexibility than is possible with a programmable exponent bias approach - the scaling factor can take on any real value (typically represented in higher precision), while programmable bias is equivalent to allowing only powers of 2 as scaling factors.

## 4 Empirical Results

Training experiments were carried out with simulated FP8 - tensor values were clipped to only those that could be represented in FP8 (including the scaling factor application and saturation). For example, prior to matrix multiplication for a fully-connected layer, both the incoming activations and the weight tensors were converted to FP8 and back to the wider representation (either FP16 or bfloat16). Arithmetic was carried out using the wider representation for two reasons: the interchange format is the focus of this paper since different processors may choose different vector- and matrix-instruction implementations, emulation of arithmetic not supported in hardware would be prohibitively slow for training of the large models. Obtaining results for large models is imperative as previous studies have identified different numerical behavior for models of different sizes (for example, R18 and R50 in [12]).

### 4.1 Training

In the FP8 training experiments we retain the same model architectures, weight initializations, and optimizer hyper-parameters as are used for higher-precision baseline training sessions. Baselines were trained in either FP16 or bfloat16, which have been shown to match the results of single-precision training sessions [14, 9]. In this study we focused on the input tensors for math-intensive operations - convolutions and matrix multiplies, to which we'll refer as GEMM-operations as they involve dot-product computations. Thus, unless otherwise specified, we clip to FP8-representable values the activation, weight, and activation gradient tensors that are inputs to GEMMs. Output tensors were left in higher precision as they typically are consumed by non-GEMM operations, such as a non-linearities or normalizations, and in a number of cases get fused with the preceding GEMM operation. Moving more tensors to FP8 is the subject of future study.

Table 2: Image Classification Models, ILSVRC12 Validation Top-1 Accuracy

Model	Baseline	FP8
VGG-16	71.27	71.11
VGG-16 BN	73.95	73.69
Inception v3	77.23	77.06
DenseNet 121	75.59	75.33
DenseNet 169	76.97	76.83
Resnet18	70.58	70.12
Resnet34	73.84	73.72
Resnet50 v1.5	76.71	76.76
Resnet101 v1.5	77.51	77.48
ResNeXt50	77.68	77.62
Xception	79.46	79.17
MobileNet v2	71.65	71.04
DeiT small	80.08	80.02

Results for image classification task are listed in Table 2. All networks were trained on ImageNet ILSVRC12 dataset, top-1 accuracy was computed on the validation dataset. All GEMM operations' inputs were clipped to FP8, including the first convolution and the last fully-connected layer, which were left in higher precision by previous studies [12, 22]. DeiT [21] is a Transformer-based architecture, the rest of the models are CNNs. With the exception of MobileNet v2, accuracy achieved by FP8 training is within run-to-run variation of higher-precision training (run-to-run variation in achieved accuracy is observed when running training sessions initialized with different random seeds). We continue work on recovering the remaining accuracy for MobileNet v2.

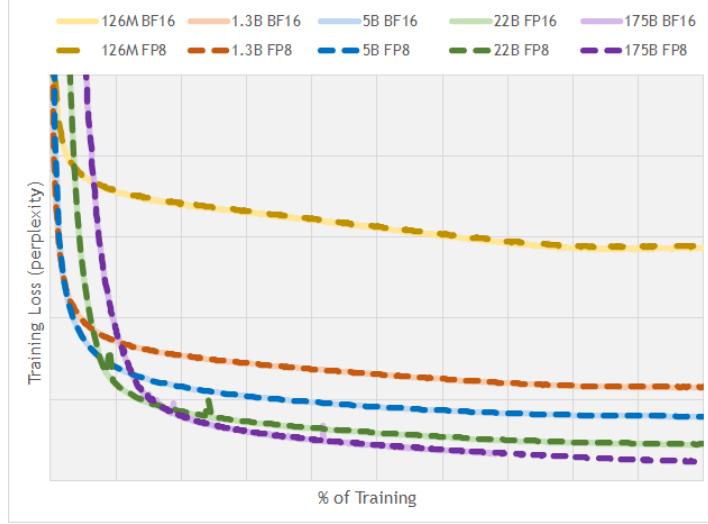


Figure 1: Training loss (perplexity) curves for various GPT-3 models. x-axis is normalized number of iterations.

Language translation task was tested using both Transformers and LSTM-based recurrent GNMT neural network. Even though Transformer-based translation models have superseded RNNs in practice, we include GNMT to more completely cover model architecture types, as well as a proxy for other tasks that still use recurrent networks. Models were trained on the WMT 2016 English->German dataset, evaluated using sacreBLEU on newstest2014 data (higher BLEU scores are better). Evaluation scores for FP8-trained models are within run-to-run variation bounds when compared to the baseline training sessions.

Table 3: Language Translation Models, English->German, BLEU Scores

Model	Baseline	FP8
GNMT	24.83	24.65
Transformer Base	26.87	26.83
Transformer Large	28.43	28.35

Training losses (perplexity, lower is better) for a variety of language models are listed in Table 4. Transformer models were trained on the Wikipedia dataset. GPT models were trained on a variant of The Pile dataset [6], augmented with Common Crawl and Common Crawl-derived datasets, as described in Section 3 of [18]. As was seen with image networks, training results of the FP8 sessions is within run-to-run noise of 16-bit training sessions. Note that 175B parameter model perplexity is reported at 75% training, as the bfloat16 baseline run has not yet completed. The FP8 training session has completed and its loss curve is consistent with successful training as shown in Figure 1. As with the vision and language translation models, we conclude that FP8 training results match those of 16-bit training sessions.

Table 4: NLP Models, Perplexity

Model	Baseline	FP8
Transformer-XL Base	22.98	22.99
Transformer-XL Large	17.80	17.75
GPT 126M	19.14	19.24
GPT 1.3B	10.62	10.66
GPT 5B	8.94	8.98
GPT 22B	7.21	7.24
GPT 175B	6.65	6.68

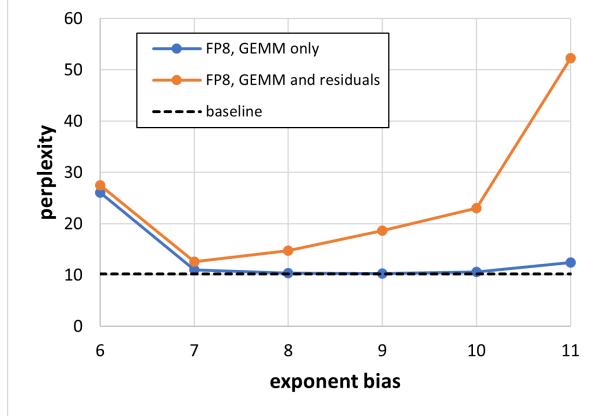


Figure 2: 1.3B GPT3 perplexity when bfloat16-trained model weight and activation input tensors are cast to E4M3 format with various exponent biases, no per-tensor scaling.

## 4.2 Inference

8-bit inference deployment is greatly simplified by FP8 training, as inference and training use the same datatypes. This is in contrast to int8 inference with networks trained in 32- or 16-bit floating point, which require post-training quantization (PTQ) calibration and sometimes quantization-aware training (QAT) in order to maintain model accuracy. Furthermore, even with quantization aware training some int8-quantized models may not completely recover the accuracy achieved with floating point [1].

We evaluate FP8 post-training quantization of models trained in 16-bit floating point. Table 5 lists inference accuracies for FP16-trained models quantized to either int8 or E4M3 for inference. Both quantizations use per-channel scaling factors for weights, per-tensor scaling factors for activations, as is common for int8 fixed-point. All input tensors to matrix-multiply operations (including attention batched matrix multiplies) were quantized. Max-calibration (choosing the scaling factor so that the maximum magnitude in a tensor is represented) is used for weights, activation tensors are calibrated using the best calibration chosen from max, percentile, and MSE methods. BERT language model evaluation on Stanford Question Answering Dataset shows that FP8 PTQ maintains accuracy while int8 PTQ leads to a significant loss of model accuracy. We also tried casting the tensors to FP8 without applying a scaling factor, which resulted in a significant accuracy loss, increasing the perplexity to 11.0. Evaluation of GPT models on wikitext103 dataset shows that while FP8 PTQ is much better at retaining model accuracy compared to int8.

Table 5: Post training quantization of models trained in 16-bit floating point. For F1 metrics higher is better, for perplexity lower is better. Best 8-bit result is bolded.

Model	Dataset (metric)	16-bit FP	int8	E4M3
BERT Base	SQuAD v1.1 (F1)	88.19	76.89	<b>88.09</b>
BERT Large	SQuAD v1.1 (F1)	90.87	89.65	<b>90.94</b>
GPT3 126M	wikitext103 (perplexity)	19.01	28.37	<b>19.43</b>
GPT3 1.3B	wikitext103 (perplexity)	10.19	12.74	<b>10.29</b>
GPT3 6.7B	wikitext103 (perplexity)	8.51	10.29	<b>8.41</b>

## 4.3 Per-tensor scaling factors

While training and inference for a number of networks can be successfully carried out in FP8 with the same scaling factor for all the tensors of the same type (in other words, choosing a single exponent bias could be possible), there are cases where per-tensor scaling factors are needed to maintain accuracy. This need becomes more pronounced when we store more of the tensors in FP8, not just inputs for GEMM operations. Figure 2 shows the FP8 inference perplexity (measured on wikitext103 dataset), when using post-training-quantization of a bfloat16-trained network. No calibration was done, weight and activation tensors were cast from bfloat16 to E4M3 type with the corresponding exponent bias. As we can see, when casting to FP8 only the inputs to GEMM operations (both weighted GEMMs as wells as two attention batched matrix multiplies that involve only activations), several exponent bias choices in the [7, 10] range lead to results matching the bfloat16 baseline. However, if we also quantize to FP8 the residual connections (input tensors for the

Add operations, which further reduces pressure on both storage and memory bandwidth) then no single exponent bias value leads to sufficient accuracy - even exponent bias of 7 results in perplexity of 12.59 which is significantly higher (worse) than 10.19 for the bfloat16 baseline. However, if instead we calibrate the tensors to have their own scaling factors (following the convention of int8 quantization to use per-channel and per-tensor scaling factors for weights and activations, respectively [23]) we achieve 10.29 and 10.44 perplexities for GEMM-only and GEMM+residuals FP8 inference.

## 5 Conclusions

In this paper we propose an FP8 binary interchange format, consisting of E4M3 and E5M2 encodings. By minimally deviating from IEEE-754 conventions for binary encoding of floating point values, we ensure that that software implementations can continue to rely on such IEEE FP properties as ability to compare and sort values using integer operations. The primary motivator for the format is acceleration of Deep Learning training and inference, by enabling smaller and more power efficient math pipelines as well as reducing memory bandwidth pressure. We demonstrate that a wide variety of neural network models for image and language tasks can be trained in FP8 to match model accuracy achieved with 16-bit training sessions, using the same model, optimizer, and training hyperparameters. Using FP8 not only accelerates and reduces resources required to train, but also simplifies 8-bit inference deployment by using the same datatypes for training and inference. Prior to FP8 8-bit inference required calibrating or fine-tuning for int8 models trained in floating point, which added complexity to the deployment process and in some cases failed to maintain accuracy.

## References

- [1] Michael J. Anderson, Benny Chen, Stephen Chen, Summer Deng, Jordan Fix, Michael Gschwind, Aravind Kalaiah, Changkyu Kim, Jaewon Lee, Jason Liang, Haixin Liu, Yinghai Lu, Jack Montgomery, Arun Moorthy, Nadathur Satish, Sam Naghshineh, Avinash Nayak, Jongsoo Park, Chris Petersen, Martin Schatz, Narayanan Sundaram, Bangsheng Tang, Peter Tang, Amy Yang, Jiecao Yu, Hector Yuen, Ying Zhang, Aravind Anburai, Vandana Balan, Harsha Bojja, Joe Boyd, Matthew Breitbach, Claudio Caldato, Anna Calvo, Garret Catron, Sneha Chandwani, Panos Christeas, Brad Cottel, Brian Coutinho, Arun Dalli, Abhishek Dhanotia, Oniel Duncan, Roman Dzhabarov, Simon Elmira, Chunli Fu, Wenyin Fu, Michael Fulthorp, Adi Gangidi, Nick Gibson, Sean Gordon, Beatriz Padilla Hernandez, Daniel Ho, Yu-Cheng Huang, Olof Johansson, Shishir Juluri, and et al. First-generation inference accelerator deployment at facebook. *arxiv*, abs/2107.04140, 2021.
- [2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.
- [3] Neil Burgess, Jelena Milanovic, Nigel Stephens, Konstantinos Monachopoulos, and David Mansell. Bfloat16 processing for neural networks. In Martin Langhammer Sylvie Boldo, editor, *26th IEEE Symposium on Computer Arithmetic, ARITH 2019, Kyoto, Japan, June 10-12, 2019*, pages 88–91. IEEE, 2017.
- [4] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.
- [5] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in neural information processing systems*, 28, 2015.

- [6] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.
- [7] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in neural information processing systems*, 29, 2016.
- [8] Jeff Johnson. Rethinking floating point for deep learning. *CoRR*, abs/1811.01721, 2018.
- [9] Dhiraj D. Kalamkar, Dheevatsa Mudigere, Naveen Mellemundi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharmendra Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhishek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of BFLOAT16 for deep learning training. *arxiv*, abs/1905.12322, 2019.
- [10] Andrey Kuzmin, Mart Van Baalen, Yuwei Ren, Markus Nagel, Jorn Peters, and Tijmen Blankevoort. FP8 quantization: The power of the exponent. *arXiv*, 2208.09225, 2022.
- [11] Edward H. Lee, Daisuke Miyashita, Elaina Chai, Boris Murmann, and S. Simon Wong. Lognet: Energy-efficient neural networks using logarithmic computation. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2017, New Orleans, LA, USA, March 5-9, 2017*, pages 5900–5904. IEEE, 2017.
- [12] Naveen Mellemundi, Sudarshan Srinivasan, Dipankar Das, and Bharat Kaul. Mixed precision training with 8-bit floating point, 2019.
- [13] Paulius Micikevicius. Mixed precision training: theory and practice, 2018. <https://on-demand.gputechconf.com/gtc/2018/presentation/s8923-training-neural-networks-with-mixed-precision-theory-and-practice.pdf>, Accessed on 2022-09-11.
- [14] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. *arxiv*, 1710.03740, 2017.
- [15] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *CoRR*, abs/1603.01025, 2016.
- [16] Badreddine Noune, Philip Jones, Daniel Justus, Dominic Masters, and Carlo Luschi. 8-bit numerical formats for deep neural networks. *arXiv preprint arXiv:2206.02915*, 2022.
- [17] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 525–542, Cham, 2016. Springer International Publishing.
- [18] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zheng, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using deepspeed and megatron to train megatron-turing NLG 530b, A large-scale generative language model. *arXiv*, 2201.11990, 2022.
- [19] Dusan Stosic and Paulius Micikevicius. Accelerating ai training with nvidia tf32 tensor cores, 2021. <https://developer.nvidia.com/blog/accelerating-ai-training-with-tf32-tensor-cores/>, Accessed on 2022-09-4.
- [20] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi (Viji) Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.
- [21] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jegou. Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning*, volume 139, pages 10347–10357, July 2021.
- [22] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.
- [23] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation. *arxiv*, 2004.09602, 2020.

- [24] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. *European conference on computer vision (ECCV)*, pages 365–382, 2018.
- [25] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuhui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.
- [26] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint*, 1606.06160, 2016.

# LIBSHALOM: Optimizing Small and Irregular-shaped Matrix Multiplications on ARMv8 Multi-Cores

Weiling Yang<sup>1,§</sup>, Jianbin Fang<sup>1,§</sup>, Dezun Dong<sup>1,\*</sup>, Xing Su<sup>1</sup>, Zheng Wang<sup>2</sup>

<sup>1</sup> College of Computer Science, National University of Defense Technology, China

<sup>2</sup> School of Computing, University of Leeds, United Kingdom

{w.yang, j.fang, dong, xingsu}@nudt.edu.cn, z.wang5@leeds.ac.uk

## ABSTRACT

General Matrix Multiplication (GEMM) is a key subroutine in high-performance computing. While the mainstream linear algebra libraries can deliver high performance on large and regular-shaped GEMM, they are inadequate for optimizing small and irregular-shaped GEMMs, which are commonly seen in new HPC applications. Some of the recent works in this direction have made promising progress on x86 architectures and GPUs but still leave much room for improvement on emerging HPC hardware built upon the ARMv8 architecture. We present LIBSHALOM, an open-source library for optimizing small and irregular-shaped GEMMs, explicitly targeting the ARMv8 architecture. LIBSHALOM builds upon the classical *Goto* algorithm but tailors it to minimize the expensive memory accessing overhead for data packing and processing small matrices. It uses analytic methods to determine GEMM kernel optimization parameters, enhancing the computation and parallelization efficiency of the GEMM kernels. We evaluate LIBSHALOM by applying it to three ARMv8 multi-core architectures and comparing it against five mainstream linear algebra libraries. Experimental results show that LIBSHALOM can consistently outperform existing solutions across GEMM workloads and hardware architectures.

## CCS CONCEPTS

- Computer systems organization; • Software and its engineering → Compilers;

### ACM Reference Format:

Weiling Yang<sup>1,§</sup>, Jianbin Fang<sup>1,§</sup>, Dezun Dong<sup>1,\*</sup>, Xing Su<sup>1</sup>, Zheng Wang<sup>2</sup>. 2021. LIBSHALOM: Optimizing Small and Irregular-shaped Matrix Multiplications on ARMv8 Multi-Cores. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3458817.3476217>

## 1 INTRODUCTION

General matrix multiplication (GEMM)<sup>1</sup> is a fundamental building block for high-performance computing (HPC) applications - from traditional scientific simulations to emerging deep learning

<sup>§</sup>Equal contribution

<sup>\*</sup>Corresponding author

<sup>1</sup>GEMM is a matrix-multiply-accumulate operation, defined as  $C = \alpha A \cdot B + \beta C$ , where  $A$  and  $B$  are matrix inputs,  $\alpha$  and  $\beta$  are scalar inputs, and  $C$  is a pre-existing matrix which is overwritten by the output. Following the naming convention of linear algebra libraries, in this work, matrix  $A$  is denoted as a  $M \times K$  matrix with  $M$  rows and  $K$  columns, matrix  $B$  is sized of  $K \times N$ , and  $C$  is sized of  $M \times N$ .

workloads. While GEMM optimization is a heavily studied field, existing linear algebra libraries mainly target GEMM operating on large matrices with regular shapes (i.e., when both dimensions of a matrix are more or less the same) [3, 7, 23, 56, 59].

Due to the diversity and the evolving nature of HPC workloads, the size and shape of the input matrices of a GEMM kernel can vary depending on the application algorithm used and input data. For example, new scientific simulation algorithms in computational fluid dynamics (CFD) like finite element methods and wave equations often adopt GEMM implementations operating on small matrices to achieve scalable performance on modern multi-core systems [28]. For example, the implementation of CP2K [28], a popular molecular dynamics simulator, extensively uses GEMMs performed on matrices of sizes  $5 \times 5$  and  $23 \times 23$ . As another example, kernels of the Nek5000 high-order solver for CFD heavily rely on GEMMs computing on  $8 \times 8$  matrices. In addition to these conventional HPC applications, new HPC workloads like deep learning and machine learning methods are often built upon small GEMM kernels [28]. Some of these data analytic algorithms also need to operate on irregular-shaped matrices [14, 32] where the magnitude of both matrix dimensions has a significant difference. For example, GEMMs used by the convolution kernels of the ResNet deep neural network [27] computes on matrices with one dimension equal to 64 while the other is greater than 3000.

These new HPC workload characteristics challenge how we optimize GEMM computation. Although the traditional linear algebra libraries like OpenBLAS [59] and BLIS [56] can deliver near-optimal performance on large and regular-shaped GEMMs, they often give poor performance on small-sized GEMMs. This is an issue reported by recent studies [28] on the x86 architecture and observed in our evaluation on ARMv8 platforms (Section 3). As we will show later in the paper, while OpenBLAS can deliver over 70% of the peak performance on large GEMMs, it gives less than 20% of the peak performance on some representative small and irregular-shaped GEMMs. As small and irregular-shaped GEMMs are now common in HPC, there is a critical need to optimize such workloads.

Recently, efforts have been made to optimize small GEMMs [28] on CPUs or irregular-shaped GEMMs on GPUs [12]. BLASFEO was among the first attempts to optimize small and irregular-shaped GEMMs within a single framework [17, 18]. While delivering promising results on x86 and GPU architectures, existing solutions are inadequate for optimizing small and irregular-shaped GEMMs on the ARMv8 based CPU architecture. As we will show in the paper, existing approaches leave much room for performance improvement on ARMv8 multi-cores due to their strategies of data packing (that maps the input matrix elements to a linear buffer), processing

edge cases of matrix elements and parallelization. Since multi-core CPUs built upon the ARMv8 architecture and instruction set are quickly emerging as an alternative to the x86-based HPC hardware [38, 44], it is highly attractive to have a library dedicated to optimizing small and irregular-shaped GEMMs on ARVm8 [52].

This paper presents LIBSHALOM<sup>2</sup>, an open-source BLAS library designed to optimize small and irregular-shaped GEMMs on ARVm8 multi-cores. As a departure from existing BLAS libraries, LIBSHALOM takes different approaches for data packing, edge-case processing and parallelization. Like mainstream BLAS libraries, LIBSHALOM builds upon the classical *Goto* GEMM algorithm [22], but it tailors this algorithm for optimizing small and irregular-shaped GEMMs on ARVm8. Unlike existing solutions that process data packing and GEMM computation in a sequential manner, LIBSHALOM leverages the SIMD instruction hide memory latency by carefully overlapping memory accesses incurred by data packing with computation operations within a GEMM kernel. Unlike conventional BLAS libraries that always apply data packing, LIBSHALOM determines, *at runtime*, if data packing is beneficial by taking into consideration the input matrix size and the GEMM computation mode. We show how simple yet effective analytic models can be developed to determine the GEMM loop tiling parameters to enhance instruction scheduling, cache locality and efficiency of parallelization and edge-case processing. We show that our analytical methods, in combination of our new, carefully optimized micro-kernel implementations, lead to significantly better performance over existing BLAS libraries when processing small and irregular-shaped GEMM on ARVm8.

**Results.** We demonstrate the benefit of LIBSHALOM by applying it to three representative ARVm8 multi-core CPUs, Phytium 2000+ [16], Kunpeng 920 [4] and ThunderX2 [34]. We evaluate LIBSHALOM on both small and irregular-shaped GEMMs as well as computation kernels from real-life applications. We compare it against five GEMM libraries that have an optimizing back-end for ARM architectures [1, 18, 28, 56, 59]. We show that LIBSHALOM consistently outperforms the competing schemes across hardware architectures, GEMM workloads, computation modes for both single-threaded and parallel executions. We showcase that, despite being a library-based approach, LIBSHALOM can outperform techniques built upon just-in-time compilation [28]. The result is a new way for implementing and optimizing GEMM kernels for small and irregular-shaped GEMMs.

**Contributions.** The technical contributions of this paper are:

- It demonstrates how the memory accessing overhead of data packing can be hidden with computations through SIMD instructions and scheduling (Section 4).
- It presents a new way to implement the GEMM computation kernels that achieves better performance over existing solutions (Section 5).
- It shows how analytical methods can be developed to determine the GEMM kernel optimization parameters for the ARVm8 architecture (Sections 4, 5 and 6).

**Online material.** We provide our code, test programs and full result data sets online at: <https://github.com/AnonymousYWL/LibShalom>.

<sup>2</sup>LIBSHALOM =Library for Small Irregular-shaped Matrix Multiplications on ARM

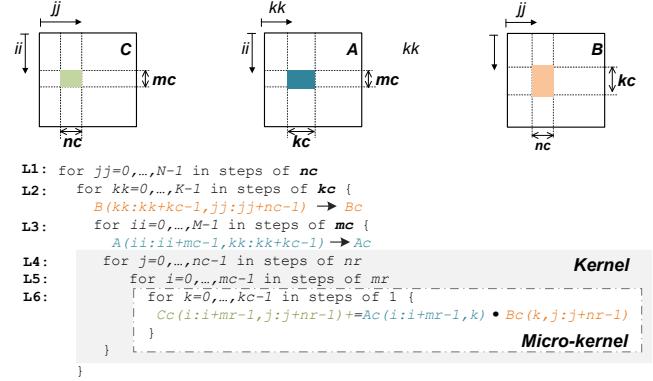


Figure 1: A typical implementation of GEMM.

## 2 BACKGROUND

### 2.1 Problem Scope

Our work focuses on optimizing GEMM performed on small and irregular-shaped matrix inputs on ARVm8 CPUs. We consider a GEMM matrix input to be small if two of its dimensions ( $M$ ,  $K$ , or  $N$ ) are of a similar size that can fit into the last-level data cache (LLC) of the CPU. By contrast, an irregular-shaped matrix is where one dimension is significantly smaller than the other, e.g., a  $64 \times 50, 176$  convolutional kernel in a deep neural network [41]. This type of matrices is also known as tall-and-skinny matrices [12, 15]. The dimensions here usually refer to  $M$  and  $N$  dimensions, and the  $K$  dimension is usually not considered [12, 39]. While recent efforts have been made to optimize small and irregular-shaped matrix multiplications, current solutions mainly target x86 architectures or GPUs. It remains unclear how small and irregular-sized GEMMs can be best optimized on emerging ARVm8 multi-core CPUs. Our work aims to close this gap.

### 2.2 General Matrix Multiply Algorithm

Figure 1 gives a high-level overview of the *Goto* GEMM algorithm [22] used by mainstream linear algebra libraries, including OpenBLAS [59] and BLIS [56]. The algorithm computes  $C = \alpha A \cdot B + \beta C$  by first partitioning and packing matrices  $A$ ,  $B$ , and  $C$  into submatrices, so that matrix multiplications can be performed on the submatrices to improve cache locality. The process of partitioning, packing and computing is performed within nested loops outlined in Figure 1, described as follows.

**Partitioning.** The outermost loop (L1) of Figure 1 groups  $C$  and  $B$  along the column direction into submatrices of sizes  $M \times nc$  and  $K \times nc$  respectively. The second level loop (L2) partitions  $A$  into submatrices on the column dimension of size  $M \times kc$ . It also further partitions the  $K \times nc$  submatrix of  $B$  into row panels of size  $kc \times nc$ . Essentially, the outermost two loops translate matrix multiplication  $A \cdot B$  to a panel-to-panel multiplication (GEPP). Then, the third level loop, L3, partitions the  $M \times kc$  panels of  $A$  into  $mc \times kc$  blocks, and partitions a  $M \times nc$  submatrix of  $C$  into row panels of size  $mc \times nc$ . The choice of  $mc$  and  $nc$  is important for maximizing the cache locality after the packing stage, described next.

**Data packing.** The outermost two loops of the GEMM algorithm packs the  $kc \times nc$  panel of  $B$  into a linear buffer,  $Bc$ . The algorithm will try the largest panel size while the entire  $Bc$  can be stored in the last level data cache [47]. Similarly, at loop L3, the algorithm packs submatrices of  $A$  generated at this loop level to a linear buffer  $Ac$  to fit into the L2 data cache. Data packing is vital for achieving high-performance GEMM by reducing memory access latency through cache locality optimization [26, 33]. However, as we will show later, the existing packing implementation is ill-suited for processing small and irregular-shaped matrices on ARMv8 multi-cores.

**Kernel.** Matrix multiplication is performed at the kernel level using a three-level loop, known as general block-times-panel multiply (GEBP) in BLAS. The kernel updates an  $mc \times nc$  panel of  $C$  by calculating the outer product (via dot to vector multiplications) of a block of  $Ac$  of size  $mc \times kc$  and a panel  $Bc$  of size  $kc \times nc$ . Specifically, the outermost loop of the kernel (L4) partitions a block  $Bc$  into *slivers* (i.e., micro-panels) of  $kc \times nr$  and the second-level loop of the kernel (L5) partitions a block  $Ac$  into slivers of  $mr \times kc$ .

**Micro-kernel.** The innermost loop of the GEBP kernel performs a sequence of updates of an  $mr \times nr$  sub-block of  $C$  using an  $mr \times 1$  column sub-sliver of  $Ac$  and a  $1 \times nr$  row sub-sliver of  $Bc$ . This innermost loop is also known as the micro-kernel in BLIS [56].

**Edge cases.** When the matrix size is not a multiple of the micro-kernel size (e.g.,  $M$  is not a multiple of  $mr$ , or  $N$  is not a multiple of  $nr$ ), we have to process the remaining elements outside the partitions. Existing approaches either pad the matrices with zeros to match the kernel size [56], or use another dedicated routine to process the remaining edge elements [59].

### 3 MOTIVATION AND OVERVIEW

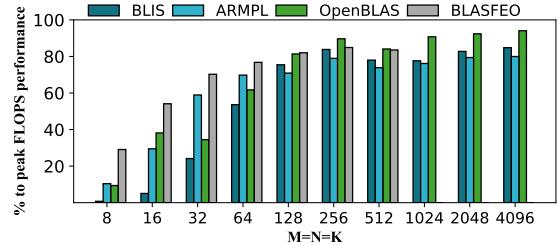
Existing GEMM libraries like OpenBLAS [59] and BLIS [56] are designed to optimize GEMM operating on large matrices. While they can obtain near peak hardware performance on large matrices, they deliver low performance on small and irregular-shaped matrices. A few libraries like BLASFEO [18] offer certain optimizations for small GEMMs but do not take full advantage of a multi-core design and the workload characteristics. As concrete examples, we evaluate the GEMM performance of four representative linear algebra libraries, OpenBLAS, BLIS, ARMPPL [1] and BLASFEO, on Phytium 2000+, a 64-core ARMv8 multi-core [11, 16, 54].

#### 3.1 Motivation Results

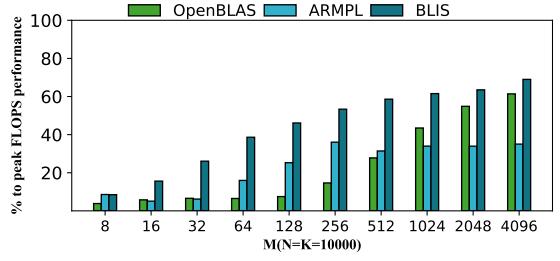
In Figure 2, we normalize the measured performances (FLOPS) to the theoretical peak CPU computational performance.

**Small-sized GEMM.** Figure 2a shows the GEMM performance on square matrices (i.e.,  $M = N = K$ ), and we see that existing GEMM libraries also give poor performance when the matrix size is small<sup>3</sup>. For example, even the best-performing library only achieves around 60% of the peak performance when the matrix size is 32. By contrast, they can achieve over 80% of the CPU peak performance when the matrix size is 256 or larger.

<sup>3</sup>Since BLASFEO is designed to optimize GEMM on matrices that can entirely fit into the L2 data cache [17, 18], it is excluded from irregular-shaped GEMM in Figure 2b.



(a) Small GEMM on Phytium 2000+



(b) Irregular-shaped GEMM on Phytium 2000+

Figure 2: GEMM performance on (a) small square and (b) irregular-shaped matrices on the ARMv8-based Phytium 2000+ processor. Existing libraries are ineffective in processing small or irregular-shaped GEMMs.

**Irregular-shaped GEMM.** Figure 2b shows the performance of irregular-shaped GEMMs when we fixed  $N$  and  $K$  to 10,000 while varying  $M$ . Matrices of this scale and size are often seen in scientific simulation kernels like high-order FEM codes and sparse direct solvers using super-block [28]. For this type of GEMM matrices, the highly optimized BLIS library can achieve 70% of the peak performance when  $M$  is 4096. However, all libraries deliver less than 40% of the peak performance when  $M$  is smaller than 128. When  $M = 16$  and  $N = 50000$ , a representative setting for certain neural network workloads [29, 30], all evaluated libraries achieve less than 25% of the theoretical CPU peak performance.

#### 3.2 Optimization Opportunities

As can be seen from the motivation results, there is much room for improvement for small and irregular-shaped GEMM on ARMv8 multi-cores, and none of the test libraries is effective at both small and irregular-shaped matrices. After close examinations, we identify three missing opportunities of current linear algebra libraries.

First, although the packing overhead is small (< 3%) on large matrices [45], it can account for 50% of the execution time for small GEMMs (e.g., when  $M = 32$  in Figure 2) and cannot be ignored. Existing GEMM libraries always pack data even when it is not beneficial to do so. When the packing overhead outweighs the benefit of small-sized GEMMs, existing solutions give a poor performance.

Secondly, we observe around 10% drop in the FLOPS when processing edge cases for small-sized matrices. This performance degradation is observed for all testing GEMM libraries, regardless of which of the two edge case strategies described in Section 2.2 is used.

While the overhead of handling edge cases is negligible for large matrices (less than 1% of the execution time when  $M = N = K = 5000$ ), the cost can be significant for small and irregular-shaped matrices.

Thirdly, we found that the existing parallelization scheme for GEMM is ineffective for irregular-shaped matrices. For example, when performing GEMM on matrices of sizes  $M = 32, N = K = 10000$ , OpenBLAS and BLIS only deliver 6% and 14% of the peak performance on Phytium 2000+. This is because when distributing the work across parallel threads, they ignore the workload characteristics of irregular-shaped GEMMs [43], creating many edge cases to be processed. These edge cases in turn bring in extra overhead that could otherwise be avoided.

### 3.3 Overview

In light of these observations, our work aims to design a better approach for packing, handling edge cases and parallelization, specifically targeting small and irregular-shaped GEMMs on the ARMv8 architecture for HPC. To this end, we develop LIBSHALOM, an open-source optimizing library for small and irregular-shaped GEMM. Following the common practice of low-level systems libraries, LIBSHALOM provides APIs in C and C++ to be used by the applications, but implements its underlying GEMM kernels in assembly for performance reasons.

**GEMM modes.** Like most BLAS libraries [1, 17, 56, 59], LIBSHALOM supports four types of GEMM kernels, NN, NT, TN and TT. Here,  $T$  and  $N$  respectively stand for a transposed and not transposed matrix. For example, GEMM for matrices  $A \cdot B$  under the NT mode means matrix  $B$  is transposed ( $T$ ) but matrix  $A$  is not ( $N$ ).

**Algorithm implementation.** Algorithm 1 outlines the LIBSHALOM’s GEMM implementation under the NN mode. Like mainstream BLAS libraries, our implementation follows the *Goto* algorithm described in Figure 1, but introduces several optimizations. Firstly, LIBSHALOM removes the always-executed packing steps, i.e., converting matrices  $B$  and  $A$  to linear buffers  $Bc$  and  $Ac$ , respectively from Figure 1. For cases that needed to be packed, we perform packing at the micro-kernel level rather than the kernel level. Secondly, we exchange the  $L2$  loop and the  $L3$  loop from Figure 1 to yield a more contiguous access on matrix  $A$ , and use loops  $L1$  and  $L3$  for parallelization (Section 6). Note that we mainly use the outer-product formulation (scalar-vector multiplication) at the micro-kernel, which has greater computation-to-memory ratio (CMR) than the inner-product formulation (vector-vector multiplication), to update matrix  $C$ . Here, the CMR is computed as the ratio of arithmetic instructions to memory load and store instructions (see Section 5.2.1). A larger CMR indicates that more arithmetic instructions are available to overlap with memory accesses to hide the memory latency.

**Roadmap.** In the following sections, we present the three key optimizations of LIBSHALOM for minimizing the overhead of small and irregular-shaped GEMMs, by redesigning the kernel (Section 4), micro-kernel (Section 5) and parallelization strategy (Section 6). Without losing generalization, we describe our approach under the NN and NT modes using single-floating point (FP32) operations. However, our optimizations are equally applicable to the other GEMM modes and double-floating points (FP64), which all are supported by LIBSHALOM. We also assume the matrices are stored in the row-major format in our discussions.

---

#### Algorithm 1: NN mode GEMM implementation

---

```

Input: Matrix A,B, Buffer Bc
Output: Matrix C
1  for jj = 0 → N step = nc do
2    for ii = 0 → M step = mc do
3      for kk = 0 → K step = kc do
4        for j = 0 → nc step = nr do
5          if size(B)>L1 then
6            for k = 0 → kc step = 1 do
7              C(ii : ii + mr, jj + j : jj + j + nr) = A(ii :
8                ii + mr; kk + k) × B(kk + k, jj + j : jj + j + nr);
9                Bc(k, 0 : nr) = B(kk + k, jj + j : jj + j + nr)
10           for i = mr → mc step = mr do
11             for k = 0 → kc step = 1 do
12               C(ii + i : ii + i + mr, jj + j : jj + j + nr) =
13                 A(ii + i : ii + i + mr, kk + k) × Bc(k, 0 : nr)
14           else
15             for i = 0 → mc step = mr do
16               for k = 0 → kc step = 1 do
17                 C(ii + i : ii + i + mr, jj + j : jj + j + nr) =
18                   A(ii + i : ii + i + mr, kk + k) × B(kk + k, jj + j
19                     jj + j + nr)

```

---

## 4 GEMM KERNEL DESIGN

### 4.1 Design Principles

For kernel computation  $A \cdot B$ , existing BLAS libraries convert each matrix to a linear buffer at the packing stage, regardless of the mode (N or T) and the matrix size. Our insight is that packing is unnecessary for small matrices or those being sequentially accessed in the micro-kernel because they can be accessed in a cache-friendly manner. For example, as matrix  $A$  is accessed rows by rows under the NN mode, the cache prefetching mechanism can largely hide the main memory access latency. For this reason, it is unnecessary to pack a large matrix  $A$  and pay the potentially expensive cost of data packing. For scenarios where packing the matrix can be profitable, LIBSHALOM tries to overlap the memory loads and stores incurred by packing with computation instructions inside the micro-kernel (Section 5.3). Therefore, LIBSHALOM only performs packing when ❶ the data cannot be accessed continuously in the micro-kernel (i.e. cache-unfriendly), or ❷ the CMR of the micro-kernel is too low to hide the memory latency without packing.

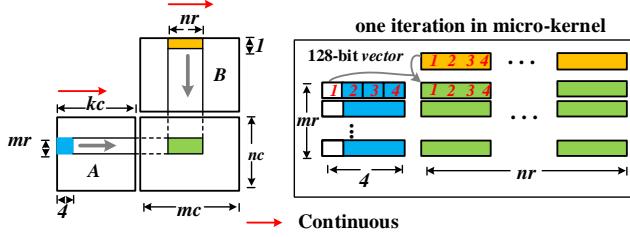
### 4.2 NN Mode Packing Strategy

Depending on the size of matrix  $B$ , we apply two packing strategies in the NN mode, described as follows.

**No packing.** If the size of matrix  $B$  is smaller than the L1 data cache (i.e.,  $\text{size}(B) < L1$  at line 12 of Algorithm 1), we skip the packing step. Instead, we go straight to divide matrix  $A$  into multiple tiles of size  $mr \times K$ , and then update matrix  $C$  at lines 13 - 15 in Algorithm 1.

**Packing large  $B$ .** If matrix  $B$  is larger than the L1 data cache capacity, we pack the tiled  $B$  into a linear buffer,  $Bc$ , and, at the same time, we update parts of matrix  $C$  in first distributed loop (lines 6–8 of Algorithm 1). Our algorithm utilizes the fused-multiply-add (FMA) instructions<sup>4</sup> to perform the outer-product computation at line 7 of Algorithm 1. As the FMA instruction can be executed concurrently with independent load and store instructions (thanks to

<sup>4</sup>The FMA instruction computes  $a \times b + c$  using one single rounding step.



**Figure 3: Kernel design using the FP32 NN kernel mode as a working example.**

out-of-order instruction scheduling), we use it to hide the packing overhead when packing matrix  $B$  at line 8 of Algorithm 1. After packing,  $Bc$  is used to update the  $mr \rightarrow mc$  rows of  $C$  during lines 9 - 11 in Algorithm 1. With a carefully designed micro-kernel (Section 5.3), we ensure that the number of CPU cycles for executing the FMA instructions can hide the overhead for filling  $Bc$ . Furthermore, since our kernel reuses  $Bc$  across computation iterations for the second distributed loop at lines 9 - 11 in Algorithm 1, we increase the chance for  $Bc$  to be kept in the L1 data cache.

**Packing choice.** For the scenarios where both matrices  $A$  and  $B$  exceed the L1 data cache size, we will pack  $B$  instead of  $A$ . Our design choice can be justified using Figure 3. From the diagram, we see that for any given row of  $A$ , the CPU can continuously access the  $0 \rightarrow kc$  elements within the same row. By contrast, the CPU can only do so for the  $0 \rightarrow nr$  elements at each row of  $B$ . Since our implementation uses a small  $nr$  (12 or 6; see Section 5) to promote the use of the vector registers, accessing to a none-packed, large  $B$  would exhibit poor cache locality. For this reason, we prioritize the packing of the matrix  $B$  at the NN mode. Because accessing matrix  $A$  is nearly continuous, we do not pack  $A$  even it is the only matrix that is larger than the L1 data cache at the NN mode.

### 4.3 Other Kernel Modes

In the NT mode, we always pack matrix  $B$  because computation is performed on the transposed (T) matrix where elements cannot be accessed along the  $N$  dimension with aligned vectorization instruction. This is depicted in Figure 3 where the continuously stored  $nr$  elements of  $B$  are transposed to be stored at discontinuous memory locations (assuming the row-major storage). The outer-product is ineffective under this setting, because this formulation requires at least one of the  $M$  dimension of  $A$  and the  $N$  dimension of  $B$  to be continuously stored in memory. To meet this requirement, LIBSHALOM chooses to pack elements from matrix  $B$  to a linear buffer  $Bc$  so that matrix elements are stored in continuous memory space. Here we also overlap computation and packing. Similarly, for the TT mode, we pack matrix  $A$  as accessing to matrix  $B$  is nearly continuous (like how we access matrix  $A$  in the NN mode). Like the NN mode kernel, we use the FMA instruction to concurrently update parts of matrix  $C$  while performing data packing.

## 5 MICRO-KERNEL DESIGN

LIBSHALOM has three types of micro-kernels, designed to minimize memory access latency and edge case processing. The first type of

---

### Algorithm 2: Main micro-kernel of all modes

---

```

// Main micro-kernel of all modes
1 for k = 0 → kc step= 4 do
2   (V0 - V6) ← A(0 : 6, k : k + 3);
3   (V7 - V9) ← Bc(*, 0 : 11);
4   (V11 - V31) ← FMA (V0 - V6)[0], (V7 - V9) /* scalar-vector multiply */
5   ...
6   (V7 - V9) ← Bc(* + 3, 0 : 11);
7   (V11 - V31) ← FMA (V0 - V6)[3], (V7 - V9)

```

---

micro-kernels is the main routine for computing  $A \cdot B$ , corresponding to lines 10-11 of Algorithm 1. The second type of micro-kernels is used at the initialization stage to perform packing while updating parts of matrix  $C$ . This corresponds to lines 6-7 of Algorithm 1 at the NN mode. The third type of micro-kernels is used to process the edge cases; see Section 2.2.

### 5.1 Design Principles

Our micro-kernel implementations aim to maximize the CMR, as prior studies have shown that optimizing this metric is important for small and irregular-shaped GEMM to achieve high-performance [28, 30]. We achieve this by taking advantages of the instruction parallelism of GEMM and the vector registers of the ARMv8 architecture, which provides 32 128-bit-wide vector registers (referred to as  $V0 - V31$ ). The key challenge here is to find the right loop tiling parameters,  $mr$  and  $nr$ , to best utilize the vector registers to maximize the CMR. To this end, we use analytic methods to determine the tiling parameters for the three types of micro-kernels, described in the next subsections. We remark that our distributed micro-kernel design is different from OpenBLAS [59] and BLIS [56] where the packing step and micro-kernel are completely separated like Figure 1.

**Working example.** In the following subsections, we describe our micro-kernel design using the FP32 NN kernel mode depicted in Figure 3 as a working example. However, our design methodology is equally applied to other kernel modes and FP64 GEMMs.

### 5.2 Main Micro-kernel

**5.2.1 Optimization constraints.** For the NN mode micro-kernel, as shown in Figure 3, we need  $mr$ ,  $nr/j$  and  $mr \times nr/j$  vector registers to store elements from matrices  $A$ ,  $B$ , and  $C$  respectively, where  $j$  is 4 and 2 for FP32 and FP64 GEMM respectively. In addition, like [47], we reserve one vector register to prefetch the elements of  $A$  or  $B$ . To make sure that the matrix elements can fit into the number of available vector registers (i.e., 31),  $mr$  and  $nr$  have to satisfy:

$$\begin{cases} mr + \frac{nr}{j} + \frac{mr \times nr}{j} \leq (32 - 1) \\ nr \% j = 0 \end{cases} \quad (1)$$

Since each vector register stores  $j$  elements, we wish to set  $nr$  to be a multiple of  $j$ , i.e.,  $nr \% j = 0$ , so that we do not waste a vector register to store fewer than  $j$  elements from matrix  $B$ .

**5.2.2 Optimization goal.** In our NN mode micro-kernel, vector registers store elements of matrix  $B$  are released after exiting from the current iteration. By contrast, vector registers for storing matrix  $A$  elements will be freed every  $j$  iterations after all the  $j$  elements on the  $K$  direction (e.g., 1, 2, 3, 4 in Figure 3 for FP32 GEMM) have been

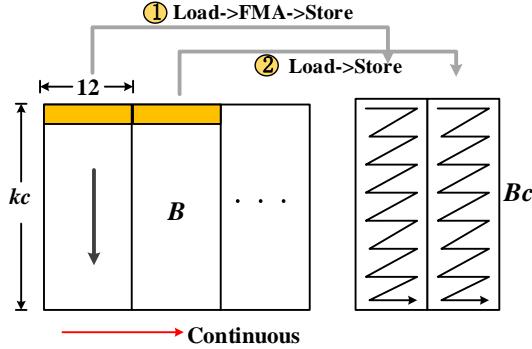


Figure 4: Packing steps of  $B$  in FP32 NN mode

used. Therefore, for every  $j$  iterations, we need  $mr$  load instructions to load elements from matrix  $A$ . Additionally, we need  $nr = nr/j \times j$  loads to fetch elements from matrix  $B$ . For computation, we apply the scalar-vector FMA instruction to  $mr \times nr$  matrix elements, which translates to  $2 \times mr \times nr$  computational operations as each FMA instruction contains two operations, addition and multiplication. Putting it together, the average CMR of our micro-kernels is:

$$CMR = \frac{2 \times mr \times nr}{mr + nr} \quad (2)$$

**5.2.3 Solving the equations.** To find an integer value of  $mr$  and  $nr$  that can maximize the CMR, we apply the Lagrange multiplier method [25] to solve the constraints defined in Equation 1 with the goal to maximize the CMR defined in Equation 2. This gives us  $mr = 7$  and  $nr = 12$  to use in our main micro-kernel implementation for the ARMv8 architecture. Not only NN mode, but we also use micro-kernel of this size for other mode GEMMs. The general process is shown in Algorithm 2.

### 5.3 Micro-kernel for Packing

The packing micro-kernel (lines 6-8 of Algorithm 1) will only be invoked if the relevant matrix is larger than the L1 data cache.

**5.3.1 Medium-sized matrix.** If matrix  $B$  is larger than the L1 data cache but smaller the LLC, we only need to pack the  $nr$  elements of  $B$  used in the current iteration of micro-kernel; after that, the elements that are continuous with these  $nr$  elements would be prefetched into the data cache.

**5.3.2 Larger and Irregular-shaped matrices.** We now describe how we pack matrices that cannot fit into the LLC cache.

**NN mode.** To reduce cache and TLB misses, when accessing the  $0 \rightarrow nr$  elements of  $B$  at the current iteration of  $j$  loop in Algorithm 1, we pack the next batch of elements of  $B$  as required by the next iteration into another part of the linear buffer,  $Bc$  in line 8 of Algorithm 1. This is because when these elements are used in the next iteration of the  $j$  loop, cache and TLB misses may occur frequently. As we iterate over this micro-kernel, we pack more elements into  $Bc$ . As a result, we will have already packed  $t \times nr$  of such elements when executing the  $t^{th}$  iteration of  $j$  loop, where  $t = 0, 1, 2 \dots n$ . Note that  $mr$  and  $nr$  in the packing kernel are set to the same values as the main kernel (i.e.,  $mr = 7$  and  $nr = 12$  for FP32). In implementation, we set  $t$  to be 0 and 1 for small and irregular-shaped GEMMs,

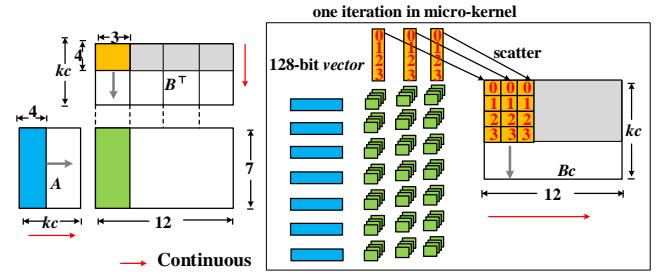


Figure 5: Micro-kernel for FP32 NT mode packing.

respectively. This means that the former only performs step 1 in Figure 4 in each iteration, while the latter performs steps 1 and 2.

**NT mode.** In this case,  $B$  is not continuous in the  $N$  dimension, which affects the use of the  $7 \times 12$  main micro-kernel to update the  $mr \rightarrow mc$  rows of  $C$ . To overcome it, we design a  $7 \times 3$  packing micro-kernel for NT mode GEMM, as show in Figure 5. In the computation process, we use the inner-product formulation to update  $C$ , and the processor accesses  $A$  and  $B$  along the  $K$  dimension. In each iteration, we use seven loads to fetch the elements of  $A$  to  $V0 - 6$ , and use three loads to fetch the elements of  $B$  to  $V7 - 9$ . The packing micro-kernel performs  $21 (7 \times 3)$  vector-vector FMAs to produce 84 ( $4 \times 21$ ) elements, which are stored in  $V10 - 31$ . At the same time, four elements of  $V7 - V9$  are scattered to  $Bc$ , and the distance between the elements is 12. Additionally, the elements in the same position of vectors are scattered to adjacent positions. For example, in Figure 5, the distance between 0 and 1 in the same vector is 12 elements, but 0 in different vectors are next to each other in  $Bc$ . At the end of the micro-kernel, the four elements of  $V10 - 31$  need to be reduced to one as using vector-vector FMAs. To get a complete  $Bc$ , we need to call packing micro-kernel four times (12/3). The micro-kernel uses the same  $7 \times kc$  tiled matrix  $A$ , but uses different  $3 \times kc$  tiled matrix  $B$ . The storage format of  $Bc$  is the same as that of Figure 4. The general process of packing micro-kernel is shown in Algorithm 3, where the vector-vector FMAs and scatter instructions occur interchangeably.

Algorithm 3: Micro-kernel for NT mode data packing

---

```

1  for  $j = 0 \rightarrow 12$  step= 3 do
2    for  $k = 0 \rightarrow kc$  step= 4 do
3       $(V0 \rightarrow V6) \leftarrow A(0 : 6, k : k + 3);$ 
4       $(V7 \rightarrow V9) \leftarrow B(0 : 2, k : k + 3);$ 
5       $(V10 - V31) \leftarrow \text{FMA } (V0 - V6), (V7 - V9) /* \text{vector-vector}$ 
        multiply
6       $(V7 - V9) \text{ scatter to } Bc(k : k + 3, j : j + 2)$ 
7      Reduce  $(V10 - v31)$  to  $V10.[0] - V31.[0];$ 
8      Store to  $C$ 

```

---

**TN and TT modes.** Following the discussion in Section 4.3, for TN mode, we apply the same strategy used for the NT mode to pack matrix  $A$ . Similarly, for TT mode, we apply the strategy used for the NN model to pack matrix  $A$ , depending on its size.

We want to highlight that our implementation interleaves the memory load and store instructions required in the packing step with FMA computation instructions like Figure 6. This is the key difference between LIBSHALOM and all existing GEMM implementations (like OpenBLAS and BLASFEO), where the packing and

ldr	s12, s13, [pB], #8	ldr	q4, [pA], #16
ldr	s14, s15, [pB], #8	ldr	q5, [pA], #16
ldr	q4, [pA], #16	ldr	q0, [PB], #16
ldr	q5, [pA], #16		
fmla	v16.4s, v4.4s, v12.s[0]	fmla	v16.4s, v4.4s, v0.s[0]
fmla	v17.4s, v5.4s, v12.s[0]	fmla	v17.4s, v5.4s, v0.s[0]
fmla	v20.4s, v4.4s, v13.s[0]	ldr	q6, [pA], #16
fmla	v21.4s, v5.4s, v13.s[0]	fmla	v20.4s, v4.4s, v0.s[1]
fmla	v24.4s, v4.4s, v14.s[0]	fmla	v21.4s, v5.4s, v0.s[1]
fmla	v25.4s, v5.4s, v14.s[0]	ldr	q7, [PA], #16
fmla	v28.4s, v4.4s, v15.s[0]	fmla	v24.4s, v4.4s, v0.s[2]
fmla	v29.4s, v5.4s, v15.s[0]	fmla	v25.4s, v5.4s, v0.s[2]
		ldr	q1, [PB], #16
		fmla	v28.4s, v4.4s, v0.s[3]
		fmla	v29.4s, v5.4s, v0.s[3]

(a)

after  
schedule

(b)

Figure 6: OpenBLAS’  $8 \times 4$  edge-case processing micro-kernel (a) and a better instruction schedule used by LIBSHALOM (b).

micro-kernel routines are executed in a sequential order. As we show later in the paper, by overlapping packing with the computation instructions, LIBSHALOM gives significantly better performance for small and irregular-shaped GEMMs.

#### 5.4 Edge Processing Micro-kernel

Our edge-case processing kernel adapts the OpenBLAS implementation [59], but enhances it with better instruction scheduling designed for small and irregular-sized GEMM. Considering the OpenBLAS  $8 \times 4$  edge micro-kernel for ARMv8 architectures shown in Figure 6a, this implementation has two drawbacks on the ARMv8 architecture. Firstly, it fails to hide the memory latency with computation instructions. That is, the load instructions are scheduled in a batch fashion. Secondly, there is no sufficiently large instruction distance between two dependent instructions. As illustrated in Figure 6b, our implementation overcomes these two drawbacks by prefetching the matrix elements required by the current iteration in the previous one and insert the load instructions between FMA instructions to hide the latency. Our experimental results show that this strategy significantly improves the OpenBLAS implementation.

#### 5.5 Other Hardware Architectures

Our approach is generally applicable and can be easily ported to other architectures. All our discussions so far target the 128-bit vector register supported by our evaluation platforms. Some new ARM-based many-cores, like the FUJITSU ARMv8-based A64FX [40] and future ARMv9 processors (e.g., NVIDIA Grace) support the latest ARM Scalable Vector Extension (SVE) [2]. This extension allows the CPU implementation to choose a vector length that is any multiple of 128 bits between 128 and 2048 bits. Our approach can be applied to a longer vector length with a revised  $mr$  and  $nr$  computed according to the available number and length of vector registers. In addition to ARM-based CPUs, our techniques can also be ported to modern x86 architectures with vectorization extensions and FMA-like instructions. Doing so will require changing the constraints of Equation 1 to match the hardware parameters to derive  $mr$  and  $nr$ . Furthermore, to adapt to different cache sizes, we can adjust the values of  $mc$ ,  $nc$  and  $kc$  [33]. Other than these parameter adjustments, we believe our analytical methods and instruction scheduling optimizations can remain unchanged.

Table 1: Hardware evaluation platform

	Phytium 2000+	KP920	ThunderX2
Peak perf. (FP32 GFLOPS)	1126.4	2662.4	1280
Number of Cores	64	64	32
Frequency	2.2 GHz	2.6 GHz	2.5 GHz
L1 cache	32KB	64KB	32KB
L2 cache	2MB	512KB	256 KB
L3 cache	None	64MB	32MB
RAM	64 GB	64 GB	64 GB

## 6 PARALLELIZATION METHODS

Small-sized GEMM is typically executed with a single thread, but irregular-shaped GEMM can benefit from parallel execution. LIBSHALOM applies a static work partitioning scheme to parallelize irregular-sized GEMM by using the *fork-join* operating system primitives. By default, we use all available cores of the CPU. For a CPU with  $T$  cores, we will spawn  $T$  parallel threads.

### 6.1 Work Partitioning

To ensure work balance among parallel threads, LIBSHALOM adopts a two-level parallelization strategy. It first divides matrix  $C$  into a grid of sub-blocks, where each thread updates one of the sub-blocks. Since we partition the work across  $T$  parallel threads, each parallel thread will perform  $\frac{M \times K}{T_m} \times \frac{N \times K}{T_n}$  computation operations for  $A \cdot B$ . Similarly, the number of memory accesses required by each parallel thread is  $\frac{M \times K}{T_m} + \frac{N \times K}{T_n}$ , where  $T_m$  and  $T_n$  are the number of threads (or cores) assigned to the  $M$  and  $N$  dimensions respectively, where  $T_m \times T_n = T$ . Therefore, the CMR for updating a sub-block is:

$$CMR = \frac{M \times N}{M \times T_n + N \times \frac{T}{T_n}} \quad (3)$$

Like our main micro-kernel design (Section 5.2), we wish to maximize the CMR. By applying the inequality of arithmetic and geometric mean method, we have:

$$CMR \leq \frac{M \times N}{2 \times \sqrt{T} \times M \times N} \quad (4)$$

where both sides of the equation will equal if  $M \times T_n = \frac{N \times T}{T_n}$ . In other words, when  $T_n = \sqrt{\frac{T \times N}{M}}$ , CMR would reach its maximum value. By taking into consideration the overhead of the packing micro-kernel, we take the up-bound value of  $T_n$ , i.e.,  $T_n = \lceil \sqrt{\frac{T \times N}{M}} \rceil$ , to maximize the CMR. We note that  $T \bmod T_n = 0$  to ensure the number of cores can be equally divided among parallel threads. For example, for parallelizing GEMM with  $M = 2048$  and  $N = 256$  on a 64-core processor, we would set  $T_n = 4$ , which leaves us with  $T_m = 16$  (as  $T_m \times T_n = T$ ). To minimize the thread synchronization overhead, we choose to parallelize two outer loops of the GEMM kernel (i.e.,  $L1$  and  $L3$  in Figure 1), instead of the inner loops [19].

## 7 EXPERIMENTAL SETUP

### 7.1 Evaluation Platforms

**Hardware.** We evaluate LIBSHALOM on three representative ARMv8 multi-core architectures: Phytium 2000+ [16], Kunpeng 920 (KP920)

[4] and ThunderX2 [34]. Table 1 lists the specification of the hardware platforms used in our evaluation. Note that on Phytium 2000+, the L2 cache is shared between a cluster of four cores, while on KP920 and ThunderX2, the L2 cache is private to a processor core.

**Systems software.** Our evaluation platforms run Linux kernel version 4.19.46. We compile the benchmarks using gcc version 8.2.1 with the “-O3” compiler option. LIBSHALOM uses OpenMP to parallelize irregular-shaped GEMMs.

## 7.2 Workloads

We evaluate LIBSHALOM by applying it to both small and irregular-sized matrices. The size ( $M \times N \times K$ ) of the small matrices ranges from  $8 \times 8 \times 8$  to  $120 \times 120 \times 120$ , which are the typical matrix sizes seen in applications like scientific simulation workloads like SeisSol [8] and Nekbox [6]. The  $M$  or  $N$  of the irregular-sized matrices used in our evaluation ranges from 32 to 256. These types of irregular-sized matrices are commonly seen in convolution neural networks (CNN) [37, 41]. Like prior work [29], we initialize the matrices by populating them with random floating-point numbers (0 to 1). In addition to the synthetic matrix inputs, we also apply LIBSHALOM to the computational kernels from CP2K (an open-source molecular dynamics simulator) [28] and the VGG CNN [30]. We report the results for running GEMM under the NN and NT modes, but we also observe similar performance trends under the TN and TT modes. More experimental results can be found on the project website [5].

## 7.3 Competitive Approaches

We compare LIBSHALOM against five GEMM libraries that have a back-end specifically tuned for ARMv8. These include OpenBLAS [59], BLIS [56] and ARMPL [1], which are designed to optimize large GEMM, as well as LIBXSMM [28] and BLASFEO [18], which specifically target small-matrix GEMM. Note that LIBXSMM uses just-in-time (JIT) compilation to optimize the GEMM kernel on the underlying architecture and uses a code cache to minimize the compilation overhead across different runs of the same kernel. Unless stated otherwise, we always run LIBXSMM on the target GEMM kernel to warm up its code cache so that the JIT compilation overhead is not included in its execution time measurement. We also note that ARMPL is the official ARM performance library, which is heavily optimized for BLAS by ARM.

## 7.4 Evaluation Methodology

For small-sized GEMM, we measure the single-threaded performance because the small matrix size does not benefit from parallel CPU execution. This is standard practice when processing small-sized matrices where parallelism is achieved by running multiple GEMM kernels to process independent matrices. For irregular-sized matrices, we report the multi-threaded performance using all the cores of a CPU. Note that because BLASFEO does not support multi-threaded execution, it is excluded from the irregular-sized matrix experiments to ensure fairness.

**Performance report.** We run each GEMM kernel 10 times and report the *geometric mean* of the runtime. We show the variations across different runs as a min-max bar.

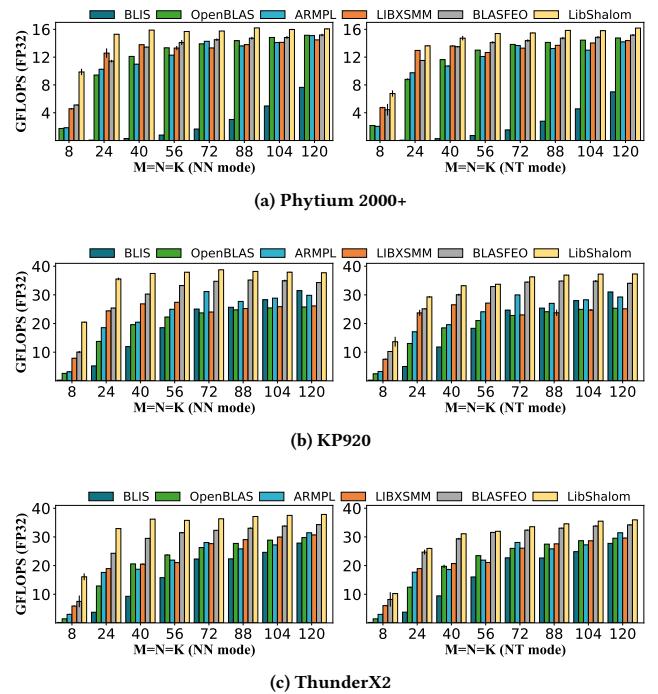


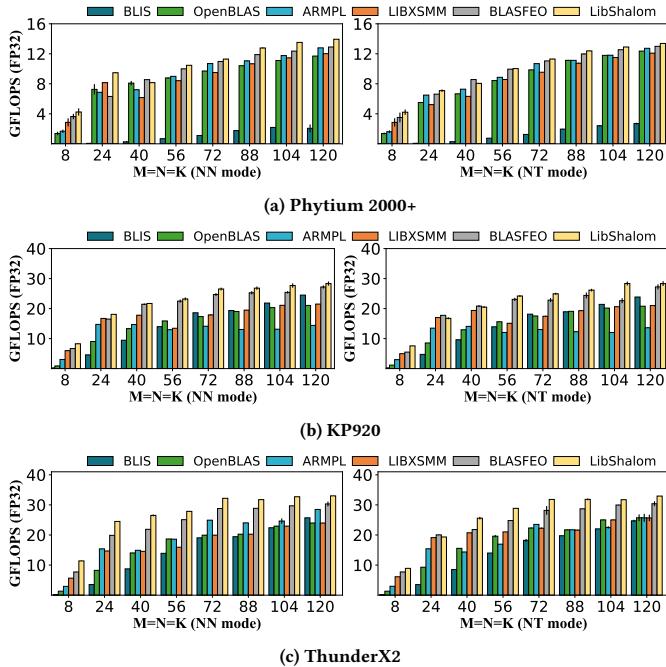
Figure 7: Small GEMMs on our evaluation platforms.

## 8 EXPERIMENTAL RESULTS

### 8.1 Single-threaded Small GEMM

In this experiment, we show the FP32 throughput for running small GEMMs on the NN and NT modes. We also observe similar trends for TN and TT modes. We also note that we obtain similar performance when applying LIBSHALOM to double-precision workloads, where the throughput is roughly half of the FP32 performance across all test methods.

Figure 7 shows the GEMM performance by first warming up the cache – a typical scenario where the *small* matrices data has been preloaded into a certain level cache before launching the GEMM kernel. This is the evaluation methodology adopted by the source publications of LIBXSMM [28] and BLASFEO [18]. In this scenario, LIBSHALOM consistently outperforms the competing methods across benchmarks and evaluation platforms. The advantage of LIBSHALOM is noticeable on smaller matrices. For example, when  $M = N = K = 8$ , LIBSHALOM delivers  $2\times$  higher throughput than BLASFEO, the best-performing alternative approach. We note that this GEMM kernel size is widely used in scientific simulation algorithms, including the NekBox CFD solver [28]. When the matrix size increases to 120, LIBSHALOM still gives at least 5% (up to 10%) higher throughput compared to the alternative approach. This benefit mainly comes from the optimized micro-kernels used by LIBSHALOM. We also observe that LIBSHALOM gives higher performance for GEMM running on the NN mode than that of the NT mode, especially on smaller sized matrices. This is because, unlike in the NT mode, NN mode GEMM under LIBSHALOM does not pack matrices that can fit into the L1 data cache; see Section 5. Overall, LIBSHALOM gives the highest throughput across the matrix settings



**Figure 8: Small GEMMs starting with a cold cache.**

by giving  $1.05 \sim 2\times$  higher throughputs across hardware platforms and manifests more significant advantages on smaller matrices.

Figure 8 shows the results when the GEMM kernel was launched from a cold cache where the matrix data are not presented in the data cache. In this evaluation scenario, LIBSHALOM demonstrates a similar performance trend as Figure 7, outperforming alternative schemes on most of the test cases. On a few matrix sizes, LIBSHALOM does not give noticeable advantages over BLASFEO - the best-performing alternative method. These matrix sizes are a (or nearly) multiple of BLASFEO's  $8 \times 8$  micro-kernel; as such, there is no or little edge-case processing overhead incurred by BLASFEO, where our edge-case optimization does not demonstrate a benefit. Nonetheless, LIBSHALOM delivers the highest overall throughput and outperforms other schemes for most of the matrix settings.

## 8.2 Parallelized Irregular-shaped GEMM

Figure 9 shows the results on irregular-shaped GEMM using all the CPU cores for parallelization on Phytium 2000+. Due to the space constraint, we show the results under the NT mode, but we observe similar performance trends in other modes. Like prior work [42], we set  $K$  to a sufficiently large number (5000 in our evaluation) to drive the last run data out of the last level data cache to avoid the artificially good performance due to a hot data cache across multiple runs. Note that we omit the results of LIBXSMM and BLASFEO in this experiment as they are tuned for small GEMMs and give a poor performance on irregular-shaped GEMMs.

LIBSHALOM significantly outperforms the alternative approaches across our evaluation platforms, yielding on average  $1.8\times$  performance improvement over the second-best performing method, BLIS. The performance benefit of LIBSHALOM tends to be more significant for smaller matrix sizes (i.e., when  $M$  or  $N$  are smaller). For

example, in Figure 9, for GEMMs with  $M = 32$ , LIBSHALOM gives  $2.6\times$  higher GFLOPS over BLIS. This is largely due to the more efficient packing strategy adopted by LIBSHALOM when processing small matrices. LIBSHALOM also demonstrates better performance over OpenBLAS and ARMPL because its parallelization strategy can minimize the overhead of processing edge cases. Although ARMPL is an official BLAS library developed by ARM for parallel GEMMs, it delivers lower performance compared to LIBSHALOM. Once again, LIBSHALOM's advantage is greater when  $M$  and  $N$  are small, suggesting that LIBSHALOM is highly effective in handling irregular-shaped matrices.

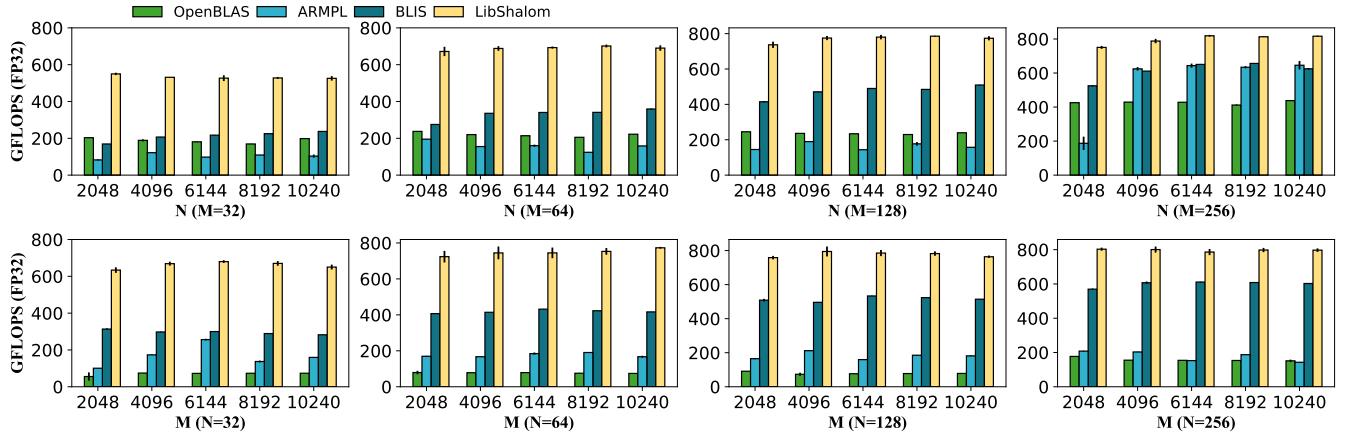
Figure 10 reports irregular-shaped GEMM performance on KP920 and ThunderX2 under the NT mode. Compared with the best-performing baseline, LIBSHALOM improves the performance by  $1.6\times$  and  $1.3\times$  respectively, on average on KP920 and ThunderX2. We also observe that LibShalom gives higher performance for irregular-shaped GEMM running on the NT mode than that of the NN mode. This is because, unlike in the NT mode, the elements of  $B$  cannot be continuously accessed along  $K$  dimension under NN mode. Overall, LIBSHALOM is generally applicable and can deliver portable performance across representative ARMv8 processors.

## 8.3 Scalability

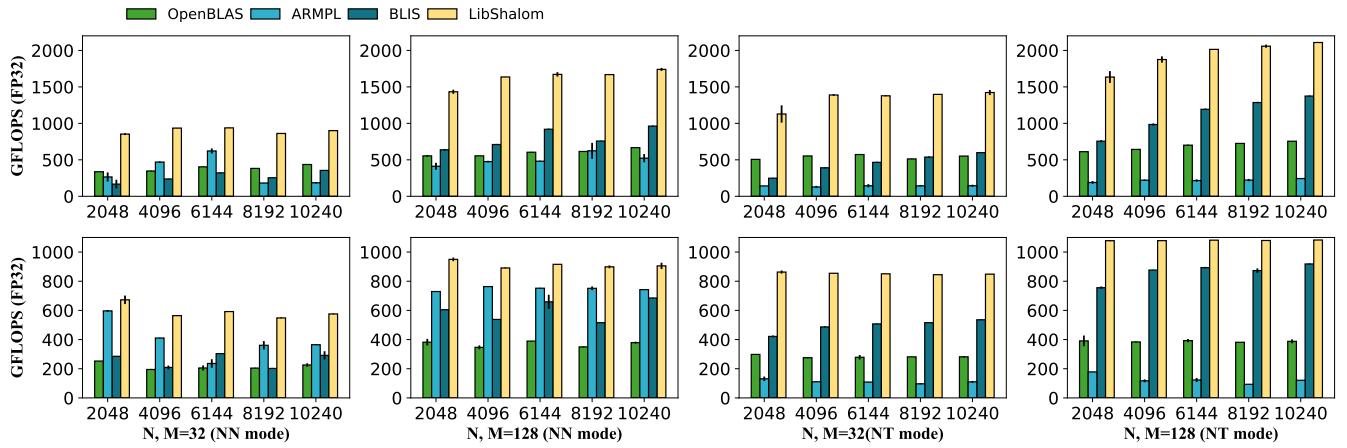
Figure 11 shows the scalability when performing an irregular-shaped GEMM kernel of  $\{M \times N \times K\} = \{64 \times 50176 \times 576\}$  from the VGG convolutional neural network [29]. The results are normalized to the performance obtained by the single-threaded OpenBLAS execution. As can be seen from the diagram, LIBSHALOM not only outperforms other approaches but also exhibits the best scalability as the number of threads used increases. And the maximum speedup is  $49\times$  for Phytium 2000+,  $82\times$  for KP920, and  $35\times$  for ThunderX2.

## 8.4 L2 Data Cache Locality

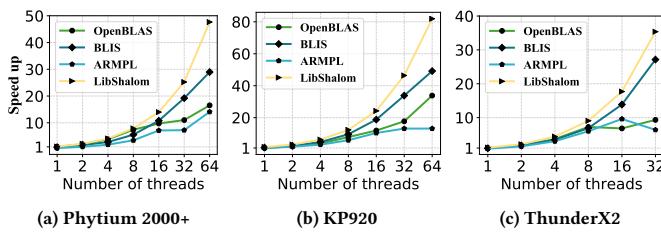
In this experiment, we measure the L2 data cache miss count using the hardware performance counter. The experiment was performed on an irregular-shaped, NT mode GEMM with input matrix sizes of  $M = 64$ ,  $N = 50176$ , where  $K$  ranging from 576 to 3744, with a step of 128. The setting ensures that the data required by the GEMM kernel can fit into the L2 data cache in an ideal scenario. Hence, a good GEMM routine should have low L2 data cache misses. The results are shown on KP920 and ThunderX2, because we can access the performance counter through the Linux perf profiler on these two platforms. Figure 12 shows the reduction of L2 cache misses using the OpenBLAS measurement as the baseline - the higher the reduction is, the better L2 cache locality an approach has. LIBSHALOM experiences the least frequent cache misses across all matrix sizes, representing around 20% reduction in the cache misses on KP920. This is because LIBSHALOM chooses to not pack matrix  $A$  under the NT mode; instead, it exchanges loops  $L2$  and  $L3$  in Figure 1 to improve the locality when accessing matrix  $A$  within the GEMM kernel. By eliminating the memory loads and stores introduced by data packing, LIBSHALOM improves the computation kernel's cache locality, leading to less frequent cache misses.



**Figure 9: Performance of irregular-shaped GEMM on Phytium 2000+ under the NT mode ( $K = 5000$ ).**



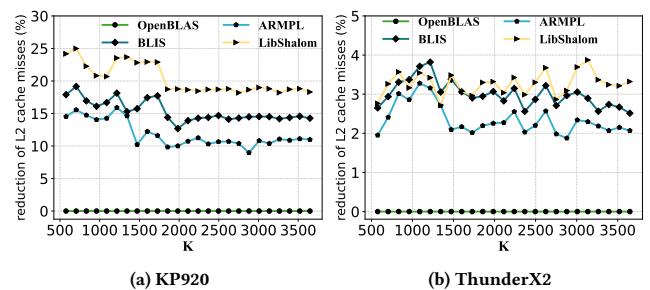
**Figure 10: Irregular-shaped GEMMs on KP920 (top row) and ThunderX2 (bottom row) under NN and NT mode ( $K = 5000$ ).**



**Figure 11: Scalability on the VGG irregular-shaped GEMM.**

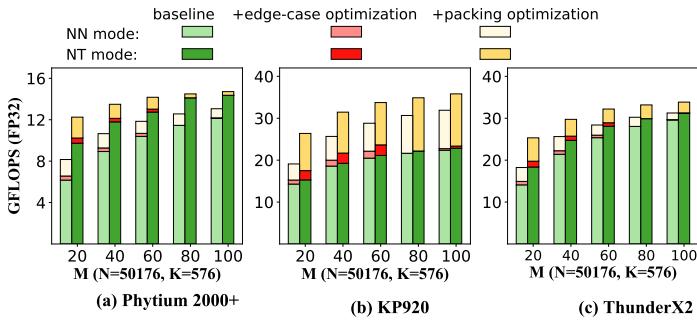
## 8.5 Breakdown of Optimization Techniques

In this experiment, we measure how the proposed packing and edge-case-processing optimizations contribute to performance improvement. Here, we use OpenBLAS as a baseline to show the speedup contribution brought by each of the two optimization techniques. The experiment was performed on single-threaded irregular-shaped, NT mode GEMM. In the experiment, we fix  $N$  and  $K$  to the VGG CNN kernel size of 50,176 and 576, but we vary  $M$  from 20 to 100 with a step of 20.



**Figure 12: Reduction of the L2 data cache misses over OpenBLAS for irregular-shaped NT mode GEMMs.**

As can be seen from Figure 13, our data packing optimization can have a significant contribution to performance improvement because this technique can overlap the expensive memory accesses with computation through FMA instructions. Our optimizations also demonstrate various degrees of benefits on different architectures. When  $M = 20$ , our two optimizations give a 1.25x and 1.6x improvement on Phytium 2000+ and KP920 respectively. The reason



**Figure 13: Breakdown of Optimizations on single-threaded irregular-shaped GEMM on three platforms.**

for the more noticeable advantage on KP920 over Phytium 2000+ is described as follows. KP920 runs a higher clock frequency over Phytium 2000+ (2.6 GHz vs 2.2 GHz), and it has more FMA units, allowing KP920 to execute more arithmetic instructions per time unit. In other words, KP920 requires more intensive FMA instructions than Phytium 2000+ to keep FMA units busy. It is more difficult for OpenBLAS’ implementation strategy to achieve a good CMR to hide the memory latency on a faster CPU where our approach gives stronger benefit.

## 8.6 Evaluation on Application Kernels

In this evaluation, we apply the tested methods to the GEMM kernels extracted from real-life application workloads. In the first experiment, we apply each approach to the FP64 small GEMM kernels from the CP2K simulation package [9]. The matrix sizes involved range between 4 – 32 [28]. As can be seen from Figure 14, LIBSHALOM gives the best performance across matrix sizes and evaluation platforms. Once again, LIBSHALOM demonstrates noticeable advantages when the input matrices are small. For example, it gives up to 2× improvement over LIBXSMM for the input matrix sizes ( $M \times N \times K$ ) is  $5 \times 5 \times 5$ . Considering LIBXSMM uses a just-in-time compilation back-end to aggressively optimize the GEMM computation, this is an impressive improvement obtained by LIBSHALOM as a library-based solution.

In the second experiment, we evaluate the irregular-shaped GEMM performance using the typical FP32 convolutional kernels from the widely-used VGG16 image classification network [29, 30]. Like prior work [29], we consider five convolution layers from VGG16, namely conv1.2, conv2.2, conv3.3, conv4.2 and conv5.2 of VGG16. These kernels perform GEMM on matrice sizes of  $M = \{64, 128, 256, 512, 512\} \times N = \{50176, 12544, 3136, 784, 196\} \times K = \{576, 1152, 2304, 4608, 4608\}$ , where the  $N$  dimension of the matrices are significantly larger than  $M$ . In this experiment, we use all the CPU cores to execute a GEMM kernel. The results are given in Figure 15. Once again, LIBSHALOM consistently outperforms all alternative approaches across GEMM kernels and evaluation platforms. LIBSHALOM’s advantage is significant for certain kernels, like conv1.2 and conv5.2, where it improves the second-best-performing approach by up to 1.6x.

## 9 RELATED WORK

High-performance linear algebra libraries are a vital component of the HPC systems software stack. A range of linear algebra or so-called BLAS libraries were developed to optimize the execution of linear algebra kernels, including GEMMs [1, 3, 56, 59]. Most of these BLAS libraries are designed to optimize GEMM operating on large and regular-shaped matrices.

Recent studies report that many new HPC workloads use small GEMM to exploit fine-grained parallelism for better scalability [17, 28]. Other works also highlight the importance of optimizing irregular-shaped GEMMs seen in machine learning workloads [29, 36]. Recent works target optimizing small GEMMs on x86 [28] or irregular-shaped GEMM on GPUs [12, 39]. Some of these techniques have been integrated into deep learning frameworks [20, 21]. However, as we have shown in the paper and our prior work [52], existing approaches give a sub-optimal performance on the ARMv8 architecture, leaving much room for improvement.

BLASFEO is designed to optimize both small and irregular-shaped GEMMs. It provides two optimization routines [17, 18]. The first covert the input matrices to the panel-major format to improve the cache locality. The second selectively packs the input matrices based on some pre-defined heuristics. For example, it does not pack a small matrix  $A$ . Likes existing BLAS libraries, BLASFEO performs data packing and computation sequentially. It also does not support parallel execution of irregular-shaped GEMMs. LIBSHALOM overcomes these limits by overlapping the memory access instructions introduced by data packing with computations and provides a highly optimized kernel for parallel execution.

LIBXSMM uses JIT code compilation technology to generate assembly code for small GEMMs [28]. This technique allows aggressive instruction-level optimization. LIBXSMM uses code cache to reuse the compilation results to reduce the overhead of JIT. However, it is designed to optimize tiny GEMM kernels where  $(MNK)^{1/3} <= 64$ . We empirically show that LIBXSMM is ineffective for optimizing the commonly used small GEMMs where the matrix sizes do not fit its design scope. Our experimental results show that despite being a library-based approach, LIBSHALOM is able to outperform LIBXSMM across GEMM workloads and evaluation platforms.

## 10 CONCLUSIONS AND FUTURE WORK

We have presented LIBSHALOM, an open-source library for optimizing small and irregular-shaped GEMMs on ARMv8 multi-cores. LIBSHALOM determines if data packing is beneficial, and when packing is deemed necessary, it uses the FMA SIMD extension to hide the non-trivial data packing overhead through instruction scheduling. We show how simple analytical models can be developed to derive the tuning parameters of a GEMM kernel. We evaluate LIBSHALOM by applying it to small and irregular-shaped GEMMs on three ARMv8 multi-core architectures and compare it against five mainstream BLAS libraries. Experimental results show that LIBSHALOM delivers consistently better overall performance across all hardware evaluation platforms.

In the future, we will look into how to extend our optimization techniques to sparse matrix computation [10, 11, 31, 55]. Another interesting direction is to open up the kernel parameters to allow

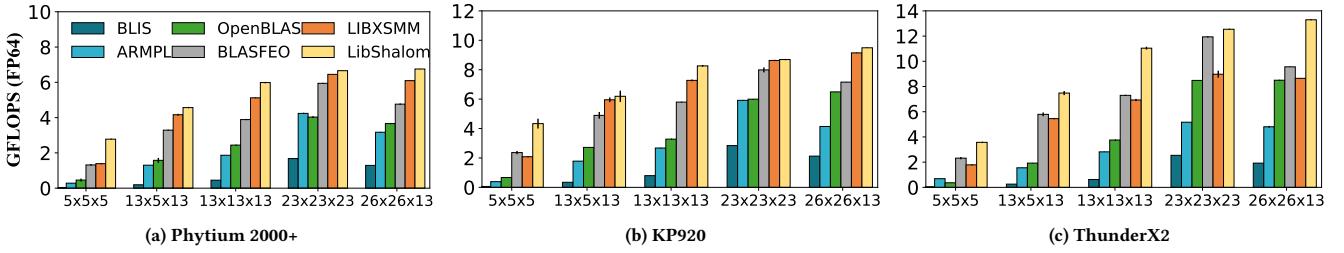


Figure 14: Small FP64 GEMM performance on CP2K kernels.

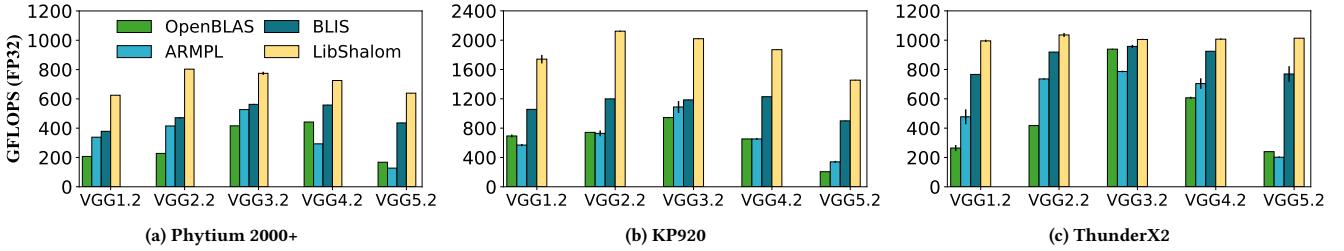


Figure 15: Performance of FP32 irregular-shaped GEMM kernels from the VGG convolutional neural network.

an auto-tuning framework to search for the optimal parameters for optimizations like instruction scheduling and compiler options [13, 24, 35, 46, 48–51, 53, 57, 58]. Our future work will consider integrating our techniques with a performance tuning framework to fine-tune the kernel parameters.

## ACKNOWLEDGMENTS

We thank Xiaoyu Wang and Yanghai Wang for their helpful discussions. This work was partially supported by the National Key R&D program of China under grant agreement 2018YFB0204300, the Excellent Youth foundation of Hunan Province under grant number 2021JJ10050, the National Natural Science Foundation of China (NSFC) under grant agreements 61972408 and 61872294, and a UK Royal Society International Collaboration grant.

## REFERENCES

- [1] [n. d.]. ARM PERFORMANCE LIBRARIES. ([n. d.]). <https://www.arm.com/products/development-tools/server-and-hpc/allinea-studio/performance-libraries>.
- [2] [n. d.]. ARMv9. ([n. d.]). <https://www.arm.com/company/news/2021/03/arm-s-answer-to-the-future-of-ai-armv9-architecture>.
- [3] [n. d.]. Intel MKL. ([n. d.]). <https://software.intel.com/en-us/mkl>.
- [4] [n. d.]. Kunpeng 920. ([n. d.]). <https://www.hisilicon.com/en/products/Kunpeng/Huawei%20Kunpeng%20920>.
- [5] [n. d.]. LibShalom. ([n. d.]). <https://github.com/AnonymousYWL/MYLIB>.
- [6] [n. d.]. Nek5000/NekBox. ([n. d.]). <https://github.com/NekBox/NekBox>.
- [7] [n. d.]. OpenCL BLAS. ([n. d.]). <https://github.com/clMathLibraries/cBLAS>.
- [8] [n. d.]. A scientific software for the numerical simulation of seismic wave phenomena and earthquake dynamics. ([n. d.]). <http://www.seissol.org/>.
- [9] Mauro Calderara, Sascha Brück, Andreas Pedersen, Mohammad H. Bani-Hashemian, Joost VandeVondele, and Mathieu Luisier. 2015. Pushing back the limit of *ab-initio* quantum transport simulations on hybrid supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, Jackie Kern and Jeffrey S. Vetter (Eds.). ACM, 3:1–3:12.
- [10] Donglin Chen, Jianbin Fang, Shizhao Chen, Chuanfu Xu, and Zheng Wang. 2019. Optimizing sparse matrix–vector multiplications on an armv8-based many-core architecture. *International Journal of Parallel Programming* 47, 3 (2019), 418–432.
- [11] Donglin Chen, Jianbin Fang, Chuanfu Xu, Shizhao Chen, and Zheng Wang. 2020. Characterizing Scalability of Sparse Matrix–Vector Multiplications on Phytium FT-2000+. *Int. J. Parallel Program.* 48, 1 (2020), 80–97.
- [12] Jieyang Chen, Nan Xiong, Xin Liang, Dingwen Tao, Sihuan Li, Kaiming Ouyang, Kai Zhao, Nathan DeBardeleben, Qiang Guan, and Zizhong Chen. 2019. TSM2: optimizing tall-and-skinny matrix–matrix multiplication on GPUs. In *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26–28, 2019*, Rudolf Eigenmann, Chen Ding, and Sally A. McKee (Eds.). ACM, 106–116.
- [13] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 219–232.
- [14] Marat Dukhan. 2019. The Indirect Convolution Algorithm. *CoRR* abs/1907.02129 (2019).
- [15] Dominik Ernst, Georg Hager, Jonas Thies, and Gerhard Wellein. 2021. Performance engineering for real and complex tall & skinny matrix multiplication kernels on GPUs. *The International Journal of High Performance Computing Applications* 35, 1 (2021), 5–19.
- [16] Jianbin Fang, Xiangke Liao, Chun Huang, and Dezun Dong. 2021. Performance Evaluation of Memory-Centric ARMv8 Many-Core Architectures: A Case Study with Phytium 2000+. *J. Comput. Sci. Technol.* 36, 1 (2021), 33–43.
- [17] Gianluca Frison, Dimitris Kouzoupis, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. 2018. BLASFEO: Basic Linear Algebra Subroutines for Embedded Optimization. *ACM Trans. Math. Softw.* 44, 4 (2018), 42:1–42:30.
- [18] Gianluca Frison, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. 2020. The BLAS API of BLASFEO: Optimizing Performance for Small Matrices. *ACM Trans. Math. Softw.* 46, 2 (2020), 15:1–15:36.
- [19] Wanrong Gao, Jianbin Fang, Chun Huang, Chuanfu Xu, and Zheng Wang. 2021. Optimizing Barrier Synchronization on ARMv8 Many-Core Architectures. In *2021 IEEE International Conference on Cluster Computing (Cluster)*.
- [20] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj D. Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of high-performance deep learning convolutions on SIMD architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11–16, 2018*. IEEE / ACM, 66:1–66:12.
- [21] Evangelos Georganas, Kunal Banerjee, Dhiraj D. Kalamkar, Sasikanth Avancha, Anand Venkat, Michael J. Anderson, Greg Henry, Hans Pabst, and Alexander Heinecke. 2020. Harnessing Deep Learning via a Single Building Block. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. New

- Orleans, LA, USA, May 18-22, 2020.* IEEE, 222–233.
- [22] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* 34, 3 (2008), 12:1–12:25.
- [23] Kazushige Goto and Robert A. van de Geijn. 2008. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.* 35, 1 (2008), 4:1–4:14.
- [24] Dominik Grewe, Zheng Wang, and Michael FP O’Boyle. 2013. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–10.
- [25] Gabriel Haeser, Oliver Hinder, and Yinyu Ye. 2021. On the behavior of Lagrange multipliers in convex and nonconvex infeasible interior point methods. *Math. Program.* 186, 1 (2021), 257–288.
- [26] Qingchang Han, Yongmin Hu, Fengwei Yu, Hailong Yang, Bing Liu, Peng Hu, Ruihao Gong, Yanfei Wang, Rui Wang, Zhongzhi Luan, and Depei Qian. 2020. Extremely Low-bit Convolution Optimization for Quantized Neural Network on Modern Computer Architectures. In *ICPP 2020: 49th International Conference on Parallel Processing, Edmonton, AB, Canada, August 17–20, 2020*, José Nelson Amaral, Lizy Kurian John, and Xipeng Shen (Eds.). ACM, 38:1–38:12.
- [27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
- [28] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13–18, 2016*, John West and Cherri M. Pancake (Eds.). IEEE Computer Society, 981–991.
- [29] Zhen Jia, Aleksandar Zlateski, Frédéric Durand, and Kai Li. 2018. Optimizing N-dimensional, winograd-based convolution for manycore CPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24–28, 2018*, Andreas Krall and Thomas R. Gross (Eds.). ACM, 109–123.
- [30] Haidong Lan, Jintao Meng, Christian Hundt, Bertil Schmidt, Minwen Deng, Xiaoning Wang, Weigu Liu, Yu Qiao, and Shengzhong Feng. 2020. FeatherCNN: Fast Inference Computation with TensorGEMM on ARM Architectures. *IEEE Trans. Parallel Distributed Syst.* 31, 3 (2020), 580–594.
- [31] Daniel Lang and Pavel Tvrđík. 2015. Evaluation criteria for sparse matrix storage formats. *IEEE Transactions on parallel and distributed systems* (2015).
- [32] Xiuhong Li, Yun Liang, Shenghen Yan, Liancheng Jia, and Yinghan Li. 2019. A coordinated tiling and batching framework for efficient GEMM on GPUs. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16–20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 229–241.
- [33] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Softw.* 43, 2 (2016), 12:1–12:18.
- [34] Filippo Mantovani, Marta García-Gasulla, José Gracia, Esteban Stafford, Fabio Banchelli, Marc Josep-Fabregó, Joel Criado-Ledesma, and Mathias Nachtmann. 2020. Performance and energy consumption of HPC workloads on a cluster based on Arm ThunderX2 CPU. *Future Gener. Comput. Syst.* 112 (2020), 800–818.
- [35] William F Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2014. Active learning accelerated automatic heuristic construction for parallel program mapping. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 481–482.
- [36] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22–26, 2020*. IEEE, 58–70.
- [37] Tran Minh Quan, David G. C. Hildebrand, and Won-Ki Jeong. 2016. FusionNet: A deep fully residual convolutional neural network for image segmentation in connectomics. *CoRR abs/1612.05360* (2016).
- [38] Nikola Rajovic, Alejandro Rico, Filippo Mantovani, Daniel Ruiz, Josep Oriol Vilarrubí, Constantino Gomez, Luna Backes, Diego Nieto, Harald Servat, Xavier Martorell, et al. 2016. The Mont-Blanc prototype: an alternative approach for HPC systems. In *SC’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 444–455.
- [39] Cody Rivera, Jieyang Chen, Nan Xiong, Jing Zhang, Shuaiwen Leon Song, and Dingwen Tao. 2021. TSM2X: High-performance tall-and-skinny matrix-matrix multiplication on GPUs. *J. Parallel Distributed Comput.* 151 (2021), 70–85.
- [40] Mitsuhsisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Oda-jima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita, and Toshiyuki Shimizu. 2020. Co-design for A64FX manycore processor and “Fugaku”. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9–19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 47.
- [41] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).
- [42] Tyler M. Smith and Robert A. van de Geijn. 2019. The MOMMS Family of Matrix Multiplication Algorithms. *CoRR abs/1904.05717* (2019).
- [43] Tyler M. Smith, Robert A. van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. 2014. Anatomy of High-Performance Many-Threaded Matrix Multiplication. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19–23, 2014*. IEEE Computer Society, 1049–1059.
- [44] Nigel Stephens. 2016. Armv8-a next-generation vector architecture for HPC. In *2016 IEEE Hot Chips 28 Symposium (HCS)*. IEEE, 1–31.
- [45] Xing Su, Xiangke Liao, Hao Jiang, Canqun Yang, and Jingling Xue. 2019. SCP: Shared Cache Partitioning for High-Performance GEMM. *TACO* 15, 4 (2019), 43:1–43:21.
- [46] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP O’Boyle. 2009. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 177–187.
- [47] Feng Wang, Hao Jiang, Ke Zuo, Xing Su, Jingling Xue, and Canqun Yang. 2015. Design and Implementation of a Highly Efficient DGEMM for 64-Bit ARMv8 Multi-core Processors. In *44th International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1–4, 2015*. IEEE Computer Society, 200–209.
- [48] Zheng Wang, Dominik Grewe, and Michael FP O’Boyle. 2014. Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 4 (2014), 1–26.
- [49] Zheng Wang and Michael FP O’Boyle. 2010. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. 307–318.
- [50] Zheng Wang and Michael FP O’Boyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.
- [51] Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael FP O’Boyle. 2014. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 1 (2014), 1–26.
- [52] Weiling Yang, Jianbin Fang, and Dezun Dong. 2021. Characterizing Small-Scale Matrix Multiplications on ARMv8-based Many-Core Architectures. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17–21, 2021*. IEEE, 101–110.
- [53] Guixin Ye, Zhanyong Tang, Huanting Wang, Dingyi Fang, Jianbin Fang, Songfang Huang, and Zheng Wang. 2020. Deep program structure modeling through multi-relational graph-based learning. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 111–123.
- [54] Xin You, Hailong Yang, Zhongzhi Luan, Yi Liu, and Depei Qian. 2019. Performance Evaluation and Analysis of Linear Algebra Kernels in the Prototype Tianhe-3 Cluster. In *Supercomputing Frontiers - 5th Asian Conference, SCFA 2019, Singapore, March 11–14, 2019, Proceedings (Lecture Notes in Computer Science)*, David Abramson and Bronis R. de Supinski (Eds.), Vol. 11416. Springer, 86–105.
- [55] Raphael Yuster and Uri Zwick. 2005. Fast sparse matrix multiplication. *ACM Transactions On Algorithms (TALG)* 1, 1 (2005), 2–13.
- [56] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3 (2015), 14:1–14:33.
- [57] Peng Zhang, Jianbin Fang, Tao Tang, Canqun Yang, and Zheng Wang. 2018. Auto-tuning streamed applications on intel xeon phi. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 515–525.
- [58] Peng Zhang, Jianbin Fang, Canqun Yang, Chun Huang, Tao Tang, and Zheng Wang. 2020. Optimizing Streaming Parallelism on Heterogeneous Many-Core Architectures. *IEEE Trans. Parallel Distributed Syst.* 31, 8 (2020), 1878–1896.
- [59] Xianyi Zhang, Qian Wang, and Yunquan Zhang. 2012. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *18th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2012, Singapore, December 17–19, 2012*. IEEE Computer Society, 684–691.

# LLM-FP4: 4-Bit Floating-Point Quantized Transformers

Shih-yang Liu<sup>\*1</sup>, Zechun Liu<sup>\*2</sup>, Xijie Huang<sup>1</sup>, Pingcheng Dong<sup>1</sup>, Kwang-Ting Cheng<sup>1</sup>

<sup>1</sup>Hong Kong University of Science and Technology, <sup>2</sup>Meta Reality Labs

{sliuau, xhuangbs, pingcheng.dong}@connect.ust.hk

zechunliu@meta.com

timcheng@ust.hk

## Abstract

We propose LLM-FP4 for quantizing both weights and activations in large language models (LLMs) down to 4-bit floating-point values, in a post-training manner. Existing post-training quantization (PTQ) solutions are primarily integer-based and struggle with bit widths below 8 bits. Compared to integer quantization, floating-point (FP) quantization is more flexible and can better handle long-tail or bell-shaped distributions, and it has emerged as a default choice in many hardware platforms. One characteristic of FP quantization is that its performance largely depends on the choice of exponent bits and clipping range. In this regard, we construct a strong FP-PTQ baseline by searching for the optimal quantization parameters. Furthermore, we observe a high inter-channel variance and low intra-channel variance pattern in activation distributions, which adds activation quantization difficulty. We recognize this pattern to be consistent across a spectrum of transformer models designed for diverse tasks, such as LLMs, BERT, and Vision Transformer models. To tackle this, we propose per-channel activation quantization and show that these additional scaling factors can be reparameterized as exponential biases of weights, incurring a negligible cost. Our method, for the first time, can quantize both weights and activations in the LLaMA-13B to only 4-bit and achieves an average score of 63.1 on the common sense zero-shot reasoning tasks, which is only 5.8 lower than the full-precision model, significantly outperforming the previous state-of-the-art by 12.7 points. Code is available at: <https://github.com/nbasy1/LLM-FP4>.

## 1 Introduction

Since the introduction of transformer architecture (Vaswani et al., 2017), transformers have superseded recursive neural networks, emerging as the dominant architecture in numerous natural language processing (NLP) tasks (Kenton and

Toutanova, 2019; Lewis et al., 2020). The transformative impact of the transformer has been further propelled by the emergence of models like GPT (Brown et al., 2020; OpenAI, 2023), catapulting the popularity of this architecture to new heights. Meanwhile, the versatility of transformers extends beyond NLP, encompassing diverse domains such as vision (Dosovitskiy et al.; Touvron et al., 2021), audio (Akbari et al., 2021), etc. This trend towards a unified architecture for different modalities represents a groundbreaking development within the realm of deep learning.

However, the advancements in transformer performance are accompanied by a corresponding increase in model size and computational costs (Kaplan et al., 2020). This poses significant challenges when attempting to leverage the full potential of transformer models in use cases where memory or computational resources are limited. Despite the extensive research and widespread adoption of transformers, the field of transformer compression remains relatively underexplored. To address this gap, our study focuses on the compression of transformers, especially through floating-point post-training quantization techniques.

Post-training quantization (PTQ) offers the advantages of simple to use with minimal fine-tuning requirements (Nagel et al., 2020; Cai et al., 2020). Existing PTQ solutions for transformers primarily focus on integer (INT) quantization (Liu et al., 2021; Yuan et al., 2022), which can be effective in certain scenarios but often break down when bit widths are below 8 bit. On the other hand, floating-point (FP) quantization has gained significant traction as a more flexible alternative, capable of better accommodating various activation and weight distributions. In fact, FP8 has emerged as the default choice in various hardware platforms, including the NVIDIA H100.

Different from integer (INT) quantization, a particular challenge in floating-point (FP) quantiza-

---

<sup>\*</sup>These authors contributed equally to this work

tion is how to select appropriate exponent bits and scale parameters. Improper parameter choices can lead to subpar or divergent quantization results. To tackle this challenge, we introduce a robust recipe for FP quantization, which leverage layer-wise reconstruction to jointly search for optimal exponent bits and maximum values. Compared to previous approaches that utilize gradient updates for exponent bits (Kuzmin et al., 2022), our search-based method proves to be more stable and consistently delivers desirable quantization results, which establishes a strong baseline for FP-PTQ.

Furthermore, our investigation uncovers an intriguing pattern of activation distributions in transformers, characterized by high inter-channel variance and low intra-channel variance. Similar patterns are also observed in previous works (Xiao et al., 2022; Dettmers et al., 2022), while we argue that this pattern is inherent to transformer architectures and not limited to specific tasks, as we have observed consistent patterns not only in large language models but also in BERT model and even vision transformers. Motivated by these findings, we introduce a novel *pre-shifted exponent bias* for FP quantization of transformers. Concretely, we leverage the per-channel activation variance computed from calibration data and reparameterize these scales as the exponential bias of the corresponding FP quantized weight vectors. This approach effectively addresses the challenge posed by high inter-channel variance while incurring negligible computational cost.

In summary, we study floating-point post-training quantization (PTQ) for transformer architectures, and the contribution of this paper includes:

- We propose a search-based framework for determining the optimal exponent bias and maximal quantization value. This method outperforms existing techniques in terms of stability and performance, establishing a strong baseline for floating-point post-training quantization.
- We propose a novel technique, *pre-shifted exponent bias*, which effectively addresses the challenge of high inter-channel variance in the transformer with negligible computational overhead.
- Experimental results demonstrate that the proposed method yields the first usable FP4 weight and activation quantized LLaMA-13B model with mere 5.8-point degradation in zero-shot reasoning tasks against the full-precision model, reducing the gap by  $\sim 70\%$  compared to the previous SoTA.

- We further extend our method to BERT and vision transformers. It surpasses the previous best 4-bit quantized BERT by 7.8 points on GLUE dataset and achieves 31.4 points higher accuracy compared to the previous SoTA ViT quantization method for 4-bit DeiT-S on ImageNet dataset.

## 2 Related Works

### 2.1 Post-Training Quantization

Model quantization can be mainly categorized into quantization-aware training (QAT) and post-training quantization (PTQ), depending on whether it involves additional training for weight fine-tuning or not. Most PTQ studies are primarily focused on convolutional neural networks (CNNs) (Nagel et al., 2020; Li et al., 2021; Wu et al., 2020; Cai et al., 2020; Nagel et al., 2019). However, with the growing popularity of transformer-based models, only a limited number of works (Bondarenko et al., 2021; Yuan et al., 2022; Ding et al., 2022) have been conducted to realize PTQ on transformers. Moreover, the existing works primarily focus on visual transformer models and exhibit inferior performance when the bit width is below 8. Therefore, in this work, we delve into the challenges of the low-bit PTQ for language transformers.

### 2.2 Floating-Point Quantization

Floating-point (FP) quantization has emerged as a promising alternative to integer quantization due to its ability to handle long-tail distributions, and offers increased flexibility (Kuzmin et al., 2022). Additionally, modern GPUs such as H100 (Micikevicius et al., 2022) now support FP quantization. Nonetheless, minimal research has been conducted on FP quantization. Only (Kuzmin et al., 2022) proposes a general FP8 quantization scheme primarily for vision tasks, and (Zhang et al., 2023) adopts a mixture of FP and INT formats quantization for LLMs. In this work, we propose FPQ baseline as a general guideline for low-bit floating-point PTQ to compress language transformer models.

## 3 Preliminaries

### 3.1 Formulation of Floating-Point Variables

A standard floating-point number is represented as:

$$X_{\text{FP}} = (-1)^s 2^{p-b} \left( 1 + \frac{d_1}{2} + \frac{d_2}{2^2} + \dots + \frac{d_m}{2^m} \right) \quad (1)$$

where  $s \in \{0, 1\}$  is the sign bit.  $d_i \in \{0, 1\}$  is  $i^{\text{th}}$  mantissa bit,  $m$  denoted number of mantissa bits.

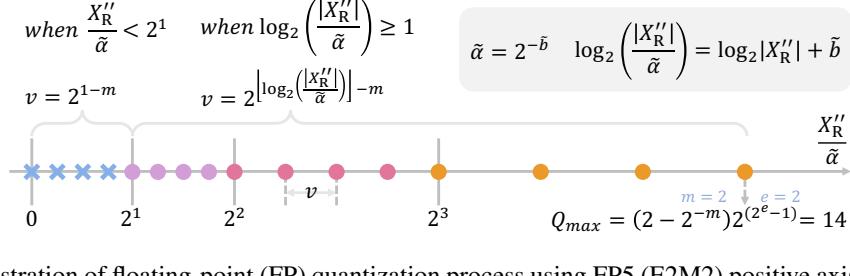


Figure 1: An illustration of floating-point (FP) quantization process using FP5 (E2M2) positive axis. The real-valued clipped  $X_R''$  in Eq. 5 is rescaled by the real-valued scaling factor  $\tilde{\alpha}$ . Then, the quantization step-size  $v$  is determined by the range  $[2^p, 2^{p+1})$  in which  $\frac{X_R''}{\tilde{\alpha}}$  falls (Eq. 9). Here,  $p \in \{0, 1, \dots, 2^{e-1}\}$  is the exponent bit value. Lastly,  $X$  can be quantized to low-bit floating point values simply by  $X_{\text{FP}} = \tilde{\alpha} \cdot v \cdot \left\lfloor \frac{X_R''}{\tilde{\alpha} \cdot v} \right\rfloor$  (Eq. 8).

$p$  is an integer in  $[0, 2^e - 1]$ , and  $e$  denotes number of exponent bits.  $b$  is an integer exponent bias. A floating point with  $j$  number exponent bits and  $k$  mantissa bits is denoted as FP format EjMk.

### 3.2 Floating-Point Quantization Process

In integer quantization, the real-valued variable  $X_R$  is quantized to an integer  $X_{\text{INT}}$  with the following formula:

$$X_{\text{INT}} = \alpha \left\lfloor \text{Clip}\left(\frac{X_R}{\alpha}, Q_{\min}, Q_{\max}\right) \right\rfloor \quad (2)$$

where  $\lfloor \cdot \rfloor$  is the rounding function.  $X_R$  is the real-valued variable,  $\alpha$  represents the full-precision scaling factor, and  $Q_{\min}, Q_{\max}$  are the min/max value of the quantization range. Similarly, a real-valued variable  $X_R$  can be converted to floating-point  $X_{\text{FP}}$  in two steps.

(1) **Scale and clip.** In FP quantization, we also scale and clip the real-valued variable before quantization as:

$$X'_R = \text{Clip}(X_R, Q_{\min}, Q_{\max}) \quad (3)$$

where the min/max value range of signed floating-point quantization can be calculated from Eq. 1:

$$Q_{\max} = -Q_{\min} = (2 - 2^{-m})2^{2^e-b-1} \quad (4)$$

Here the integer exponent bias  $b$  is another adjustable hyperparameter controlling  $Q_{\max}$  and  $Q_{\min}$ , which has similar functionality as  $\alpha$ . Therefore, for simplicity, we reformulate Eq. 3 as:

$$X''_R = \text{Clip}\left(X_R, \tilde{Q}_{\min}, \tilde{Q}_{\max}\right), \quad (5)$$

where

$$\begin{aligned} \tilde{Q}_{\max} &= \alpha Q_{\max} = \alpha \cdot (2 - 2^{-m})2^{2^e-b-1} \\ &= \alpha \cdot 2^{-b} \cdot (2 - 2^{-m})2^{2^e-0-1} \\ &= 2^{-\tilde{b}} \cdot (2 - 2^{-m})2^{2^e-0-1} \end{aligned} \quad (6)$$

Note that we combine the tensor-wise real-valued scaling factor  $\alpha$  with integer exponent bias  $b$  to form a new scaling factor  $\tilde{\alpha} = 2^{-\tilde{b}} = 2^{-b} \cdot \alpha$ . Here  $\tilde{b}$  denotes a relaxed tensor-wise real-valued exponent, and we can derive  $\tilde{b}$  from the desired clipping value  $\tilde{Q}_{\max}$  from Eq. 6 as:

$$\tilde{b} = 2^e - \log_2 \tilde{Q}_{\max} + \log_2(2 - 2^{-m}) - 1 \quad (7)$$

(2) **Compare and quantize.** Different from integer quantization, which simply utilizes the rounding function to convert the real-valued variables to quantized ones, in floating-point quantization, there is an additional step of comparing  $X''_R$  with quantization levels and then quantize:

$$X_{\text{FP}} = \tilde{\alpha} \cdot v \cdot \left\lfloor \frac{X''_R}{\tilde{\alpha} \cdot v} \right\rfloor \quad (8)$$

where  $X''_R$  is clipped real-valued variable (Eq. 5),  $\tilde{\alpha}$  is the tensor-wise floating-point scaling factor, and  $v$  is an integer power of 2.

$$v = \begin{cases} 2^{\lfloor \log_2 |X''_R| + \tilde{b} \rfloor - m} & \text{if } \lfloor \log_2 |X''_R| + \tilde{b} \rfloor \geq 1 \\ 2^{1-m} & \text{otherwise} \end{cases} \quad (9)$$

Here we select the quantization level  $v$  according to the magnitude of  $\frac{X''_R}{\tilde{\alpha}}$ , which equals to  $X''_R \cdot 2^{\tilde{b}}$ . Then the floating-point quantized variables can be derived with Eq. 8. The illustration of the quantization process is in Fig. 1, detailed explanation can also be found in (Micikevicius et al., 2022).

### 3.3 Floating-Point Matrix Multiplication

With the floating-point quantized variables, the matrix multiplication is formulated as:

$$\mathbf{O}_{\text{out}}^{i,k} = \mathbf{X}_{\text{FP}}^{i,:} \mathbf{W}_{\text{FP}}^{:,k} = \tilde{\alpha}_{\mathbf{X}} \tilde{\alpha}_{\mathbf{W}}^k \tilde{\mathbf{X}}_{\text{FP}}^{i,:} \tilde{\mathbf{W}}_{\text{FP}}^{:,k} \quad (10)$$

Here in per-tensor activation quantization and per-channel weight quantization,  $\mathbf{X}_{\text{FP}}^{i,:}$  denotes  $i^{\text{th}}$  row

in the activation matrix and  $\mathbf{W}_{\text{FP}}^{:,k}$  denotes  $k^{\text{th}}$  column in the weight matrix, such that each element  $\mathbf{O}_{out}^{i,k}$  in the output matrix is computed by the product of two real-valued scalars  $\tilde{\alpha}_x$  and  $\tilde{\alpha}_w^k$  times the corresponding quantized activation and weight vectors. We depict all the possible quantization granularity options that support such efficient matrix multiplication in Appendix D.

## 4 Method

In this section, we begin by introducing our joint format and max value search, which establishes our strong baseline and already achieves state-of-the-art results at 8-bit and 6-bit quantization. Then we present an efficient *pre-shifted exponent bias* to tackle the catastrophic high inter-channel activation variance in transformer models and push the quantization limit to 4-bit.

### 4.1 Joint Format and Max Value Search

The objective of post-training quantization is to minimize the perturbation ( $\delta \mathbf{X} = \mathbf{X}_{\text{FP}} - \mathbf{X}_{\text{R}}$ ) introduced by quantization to the pre-trained real-valued network:

$$\min \mathbb{E}[\mathcal{L}(\mathbf{X}_{\text{R}} + \delta \mathbf{X}) - \mathcal{L}(\mathbf{X}_{\text{R}})] \quad (11)$$

In this study, we adopt the setting presented in (Choukroun et al., 2019; Wu et al., 2020), which assumes a positive correlation between the change in the intermediate output of the quantized model and Eq. 11. Therefore, minimizing the distance between the intermediate output of the quantized layer ( $\hat{\mathbf{O}}$ ) and the output of the original layer ( $\mathbf{O}$ ) leads to minimize Eq. 11. Hence, the objective loss metric is formulated as:

$$\min (\hat{\mathbf{O}} - \mathbf{O})^2 \quad (12)$$

which is used to search for the optimal FP quantization function in the following proposed framework.

The challenges in FP quantization arise from its sensitivity to the quantization format and clipping range. Undesirable format selection will result in a catastrophic error rate. In addition, we observe that the optimal clipping range varies depending on the format used. Previous work (Kuzmin et al., 2022) on floating-point (FP) quantization-aware training (QAT) proposed to learn both the FP format and maximum value with gradients. However, we find this method suffers from over-fitting in PTQ, with accuracy being even worse than naïve MinMax

method, details can be found in Appendix E. Instead, we propose a search-based algorithm that jointly determines the optimal format and its associated clipping range to address this challenge.

The searching process is conducted layer by layer with the metric of minimizing Eq. 12. The output of matrix multiplication corresponding to each sub-module is denoted as  $\mathbf{O} = \mathbf{XY}$ , where  $\mathbf{Y}$  can be either a weight tensor  $\mathbf{W}$  or another activation tensor.

The search space of  $q$ -bit FP format includes all formats except for the format with an exponent bit equal to 0, as the quantization of the format with an exponent bit equal to 1 already degenerates to INT quantization. We search for the real-valued exponent bias  $\tilde{b}$ , which equals to the logarithm of the scaling factor. We initialize  $\tilde{b}_x$  and  $\tilde{b}_y$  from Eq. 7 with  $Q_{\max}$  equals the maximum value of  $|\mathbf{X}_R|$  and  $|\mathbf{Y}_R|$ , respectively. We then define the search space of  $\tilde{b}_x$  and  $\tilde{b}_y$  by linearly dividing  $[\gamma_1 \tilde{b}_x^{\text{init}}, \gamma_2 \tilde{b}_x^{\text{init}}]$  and  $[\gamma_1 \tilde{b}_y^{\text{init}}, \gamma_2 \tilde{b}_y^{\text{init}}]$  into  $k$  intervals, where  $\gamma_1$  and  $\gamma_2$  are empirically set to 0.01 and 1.2, and  $k = 100$ .

The search process is outlined in Alg.1. We search the quantization scheme in all the matrix multiplication layers in parallel following (Yuan et al., 2022; Bai et al., 2022). The algorithm can be divided into two parts. (1) Do forward propagation to store the intermediate raw output of each layer  $l$ . (2) Iteratively update the optimal format and biases for each layer for three rounds by minimizing the reconstruction metric (Eq. 12). We name this search-based framework as *Floating Point Quantization Baseline* (FPQ baseline), and it can already achieve state-of-the-art results on both 8-bit and 6-bit settings.

### 4.2 Pre-Shifted Exponent Bias

In transformer architectures, we observed an intriguing phenomenon of high inter-channel variance. As shown in Fig.2, the magnitudes of values within the same channel are close to each other but exhibit significant differences across different channels. This phenomenon is not only observed in language models (*i.e.*, LLaMA and BERT) but also significant in vision transformer models. Since outlier channels are often orders of magnitude bigger than the rest, they will dominate the quantization precision of the quantized tensor, resulting in less representation capacity for those channels with smaller magnitudes (Xiao et al., 2022). This makes tensor-wise or token-wise scaling factor insufficient for accurate activations quantization.

---

**Algorithm 1** FPQ baseline

---

- 1: **Input:** Calibration dataset, Full-precision Model  $M$ , Quantization format search space  $R_X$  (e.g.,  $R_X = \{E3M0, E2M1, E1M2\}$  for FP4), number of round  $n = 3$ ,
- 2: **Output:** FP  $q$  Quantized model
- 3: **for**  $l$  in  $1^{st}$  to  $L^{th}$  layer in  $M$  **do**
- 4:     Forward & collect raw output  $O^l = X^l Y^l$  of layer  $l$ ;
- 5: **end for**
- 6: **for**  $l$  in  $1^{st}$  to  $L^{th}$  layer in  $M$  **do**
- 7:     Initialize the FP format search space w.r.t  $X^l$  and  $Y^l$  as  $R_X = \{r_X^1, r_X^2, \dots, r_X^t\}$  and  $R_Y = \{r_Y^1, r_Y^2, \dots, r_Y^t\}$ .
- 8:     Initialize bias  $\tilde{b}_X^l, \tilde{b}_Y^l$  with Eq.7 for each format candidate  $r_X^i \in R_X$  and  $r_Y^i \in R_Y$ .
- 9:     Generate search space of  $\tilde{b}_X^l$  in  $t$  formats to be  $[\gamma_1 \tilde{b}_X^{init}, \gamma_2 \tilde{b}_X^{init}]$  and  $\tilde{b}_Y^l$  to be  $[\gamma_1 \tilde{b}_Y^{init}, \gamma_2 \tilde{b}_Y^{init}]$ .
- 10:    **for** 0 to  $n$  **do**
- 11:      Search for  $\tilde{b}_X^l$  w.r.t each  $r_X^i$  that minimizes Eq.12
- 12:      Search for  $r_X^i \in R_X$  that minimizes Eq.12
- 13:      Search for  $\tilde{b}_Y^l$  w.r.t each  $r_Y^i$  that minimizes Eq.12
- 14:      Search for  $r_Y^i \in R_Y$  that minimizes Eq.12
- 15:    **end for**
- 16: **end for**

---

However, applying per-channel scaling factors for activations poses challenges to efficient matrix multiplication, because the scaling factor is not a shared constant along the multiplication direction and cannot be extracted as Eq. 10. To address this challenge, we introduce *pre-shifted exponent bias*, which allows us to calculate per-channel scaling factors from activations. These scaling factors are then re-parameterized as the exponent biases of the corresponding weights. This method effectively handles high inter-channel variance while maintaining nearly identical efficiency to per-tensor quantization.

Recalling in Eq. 7, we extracted the tensor-wise integer exponent bias  $b$  and times it with real-valued scaling factor  $\alpha$  and becomes a new scaling factor  $\tilde{\alpha} = 2^{-\tilde{b}} = 2^{-b} \cdot \alpha$ . Then, the floating-point quantization formula in Eq. 13 becomes:

$$X_{FP} = 2^{-\tilde{b}} (-1)^s 2^{p-0} \left(1 + \frac{d_1}{2} + \frac{d_2}{2^2} + \dots + \frac{d_m}{2^m}\right) \quad (13)$$

We note that after the bias is absorbed in the scaling factor, the original bias term ( $b^{ori}$ ) in the FP formula is always zero. In dealing with the inter-channel variance, we devise an innovative usage of this integer exponent bias: we set it to be a per-channel variant ( $b^{ori} \in \mathbb{Z}^c$ ).

Then the calculation of the channel-wise integer bias vector ( $b^{ori}$ ) is very straightforward. We first calculate the initial per-channel real-valued scaling factor ( $2^{-\tilde{b}_j}$ ) from the per-channel maximum

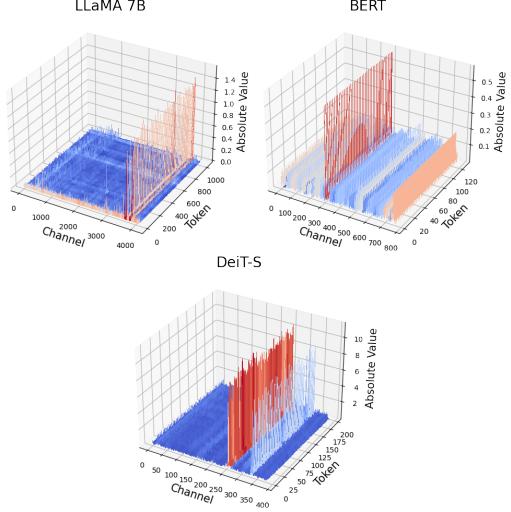


Figure 2: Magnitude of the output activations of the feed-forward network blocks in LLaMA-7B, BERT, and DeiT-S.

values:

$$\tilde{b}_j = 2^e - \log_2(\max(|\mathbf{X}_R^{:,j}|)) + \log_2(2 - 2^{-m}) - 1 \quad (14)$$

Here  $\mathbf{X}_R^{:,j}$  denotes the  $j^{th}$  channel in the activation matrix. Then we separate  $\tilde{b}$  to a tensor-wise real-valued scaling factor plus a channel-wise integer scaling factor:

$$\begin{aligned} \tilde{\mathbf{b}} &= \tilde{\rho} + \mathbf{b}^{ori} \\ &= \tilde{\rho} + \text{clip}([\tilde{\mathbf{b}} - \tilde{\rho}], 0, 2^{e-1}) \end{aligned} \quad (15)$$

where  $\tilde{\rho} \in \mathbb{R}^1$ ,  $\mathbf{b}^{ori} \in \mathbb{Z}^c$ . Then the formula for one of the entries in the  $j^{th}$  channel of  $\mathbf{X}$  can be rewrote as follows:

$$\begin{aligned} X_{FP} &= 2^{-\tilde{b}_j} (-1)^s 2^{p-0} \left(1 + \frac{d_1}{2} + \dots + \frac{d_m}{2^m}\right) \\ &= 2^{-\tilde{\rho}} (-1)^s 2^{p-\mathbf{b}_j^{ori}} \left(1 + \frac{d_1}{2} + \dots + \frac{d_m}{2^m}\right) \end{aligned} \quad (16)$$

Note that the bias  $\mathbf{b}^{ori}$  is constrained to integers within  $[0, 2^e - 1]$ , compatible with the standard floating-point number calculation. Nevertheless, adding different biases for each channel during inference may still cause some extra hardware operations. Thus, we re-parameterized the per-channel activation bias into a weight tensor and pre-computed the weights using the calibration set. This way, the exponent biases shifting only happens in the calibration stage. Then, an element in  $j^{th}$  channel of activation tensors  $X$  becomes:

$$X_{FP} = 2^{-\tilde{\rho}} (-1)^s 2^{p-0} \left(1 + \frac{d_1}{2} + \dots + \frac{d_m}{2^m}\right) \quad (17)$$

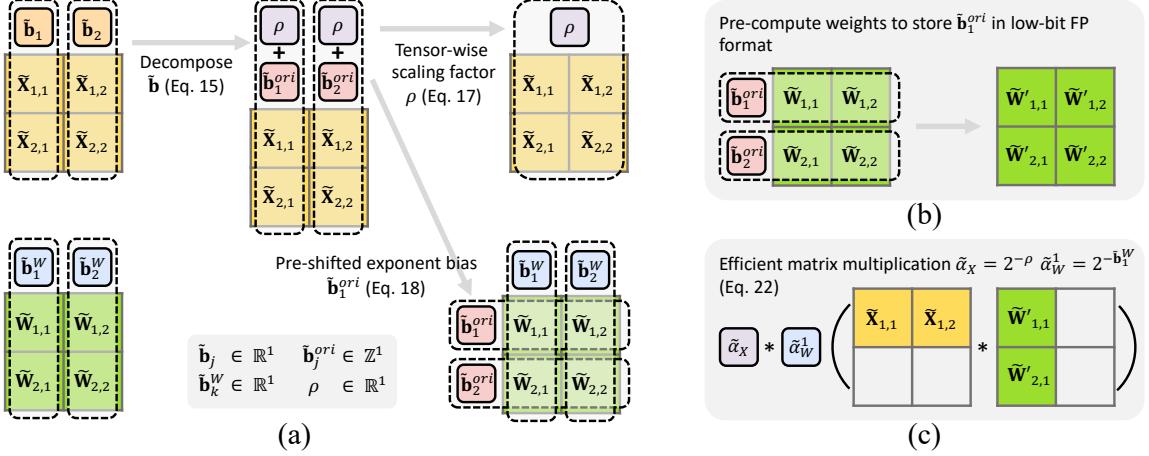


Figure 3: **Overview of pre-shifted exponent bias method:** (a) **Search phase:** The real-valued channel-wise scaling exponent bias for activations ( $\tilde{\mathbf{b}}_j$ ) is partitioned into a real-valued tensor-wise exponent bias ( $\rho$ ), and the integer-based channel-wise exponent bias ( $\tilde{\mathbf{b}}_j^{ori}$ ). (b) **Reparameterization and weight pre-computation:** Once the optimal values are determined on the calibration set,  $\tilde{\mathbf{b}}_j^{ori}$  are re-parameterized into the weight tensor. The weights are pre-computed to apply the bias, therefore this is a one-time cost. (c) **Inference phase:** The method leverages efficient matrix multiplication between low-bit floating-point matrices.

and the corresponding weight element in  $j^{th}$  row of the weight tensor  $W$  becomes:

$$W_{FP} = 2^{-\tilde{\mathbf{b}}_j^W} (-1)^s 2^{p - \tilde{\mathbf{b}}_j^{ori}} \left(1 + \frac{d_1}{2} + \dots + \frac{d_m}{2^m}\right) \quad (18)$$

As result, efficient matrix multiplication in Eq.10 is reformulated as:

$$\mathbf{O}_{out}^{i,k} = \mathbf{X}_{FP}^{i,:} \mathbf{W}_{FP}^{:,k} = \tilde{\alpha}_X \tilde{\alpha}_W^k \tilde{\mathbf{X}}_{FP}^{i,:} (\beta \odot \tilde{\mathbf{W}}_{FP}^{:,k}) \quad (19)$$

where  $\odot$  is the element-wise multiplication,  $\beta = 2^{-\tilde{\mathbf{b}}^{ori}}$  and  $(\beta \odot \tilde{\mathbf{W}}_{FP}^{:,k})$  can be pre-calculated and stored in low-bit FP format. We depict the overall *pre-shifted exponent bias* method in Fig.3. This method applies to quantizing all the fully-connected layers. During the search process, we initialize  $\tilde{\rho}_x$  as the  $\min_j(\tilde{\mathbf{b}}_j)$ . Then, we fixed  $\tilde{\mathbf{b}}_x$  to be the bias calculated from the Eq. 14 and search for the optimal  $\tilde{\rho}_x$  from  $[\gamma_1 \tilde{\rho}_x^{init}, \gamma_2 \tilde{\rho}_x^{init}]$ .

Combining *pre-shifted exponent bias* method with the joint format and max-value search framework(FPQ baseline), we name our method as (FPQ), short for *Floating Point Quantization*.

## 5 Experiments

To validate the effectiveness of the proposed method, we conduct experiments on LLaMA (Touvron et al., 2023) and BERT (Devlin et al., 2019) models in 5.2.1 and Sections 5.2.2. Further, in Section 5.2.3 we show that our method also generalizes well to vision transformer architectures. We present ablation studies on the calibration size

and search range in Section 5.3, and analyze the hardware costs of implementing FP operators in Section 5.4.

### 5.1 Experiments Details

We adopt per-tensor quantization for activation and per-channel quantization for weight. We employ layer reconstruction following the settings of (Yuan et al., 2022; Nagel et al., 2020), and parallel quantization based on the approach outlined in (Bai et al., 2022; Yuan et al., 2022). A more detailed discussion regarding our implementation decisions can be found in Appendix F. For LLaMA models, we quantize all the weight and activation tensors in fully-connected layers for a fair comparison with previous work (Xiao et al., 2022; Liu et al., 2023). For BERT and ViT models, both fully-connected layers and activation-activation multiplication tensors in the self-attention module are quantized. Note that for FPQ on BERT (Devlin et al., 2019) and ViTs models, the reconstruction metric Eq. 12 is substituted with a Hessian approximation loss metric. This substitution is further detailed in Appendix A.

### 5.2 Main Results

#### 5.2.1 LLM Zero-Shot Reasoning

We evaluate the effectiveness of FPQ for LLaMA-7B/LLaMA-13B (Touvron et al., 2023) on common sense zero-shot reasoning tasks. For the calibration data, we sample 32 random segments with 2048 tokens length from the C4 (Raffel et al., 2020)

Quant Method	#Bits (E/W/A)	# Calib	BoolQ	PIQA	HellaSwag	WinoGrande	ARC-e	ARC-c	Avg.
LLaMA-7B Full-precision	16/16/16	-	75.1	78.7	56.9	69.9	75.3	41.9	66.3
SmoothQuant ( <a href="#">Xiao et al., 2022</a> )	8/8/8	32	64.3	66.8	40.5	57.4	59.0	29.6	52.9
	8/8/8	32	74.9	78.6	56.8	69.5	75.5	41.6	<b>66.1</b>
	16/8/8	512	74.0	77.5	55.0	69.6	74.4	37.4	64.6
	8/8/8	32	75.8	78.3	55.9	69.5	75.6	41.3	<b>66.1</b>
	FPQ	32	75.6	78.2	56.6	70.2	74.6	40.7	66.0
GPTQ ( <a href="#">Frantar et al., 2023</a> )	4/4/16	32	64.1	76.1	51.6	66.3	72.4	40.0	61.7
	4/4/16	32	73.0	77.9	55.2	69.1	73.6	40.9	64.9
	4/4/16	128	73.3	77.9	54.9	67.9	72.7	37.4	64.0
	4/4/16	32	74.8	77.9	55.6	69.5	75.2	41.0	<b>65.7</b>
	FPQ	32	74.2	77.8	55.8	69.9	74.9	40.4	65.5
MinMax INT Quant	4/4/8	32	50.4	56.5	27.9	46.5	36.1	21.2	39.7
	MinMax FP Quant (E2M1/E4M3)	32	73.0	77.5	55.0	69.3	73.6	40.9	64.9
	FPQ baseline	32	75.0	77.6	55.9	69.9	74.3	39.4	65.3
	FPQ	32	75.0	77.7	55.5	69.8	74.5	39.9	<b>65.4</b>
SmoothQuant ( <a href="#">Xiao et al., 2022</a> )	4/4/4	32	54.1	51.7	25.6	49.8	24.7	22.9	38.1
	MinMax FP Quant (E2M1)	32	47.3	53.1	25.7	50.7	25.1	22.4	37.4
	16/4/4	512	54.1	62.8	41.5	52.6	50.6	32.9	49.1
	LLM-QAT ( <a href="#">Liu et al., 2023</a> )	16/4/4	(QAT)	63.5	64.3	55.6	52.9	50.3	30.2
	FPQ baseline	32	57.4	56.6	30.2	51.1	37.7	23.2	42.7
	FPQ	32	64.2	73.5	47.8	63.7	65.9	33.6	<b>58.1</b>
LLaMA-13B Full-precision	16/16/16	-	77.9	79.2	59.9	72.6	77.4	46.4	68.9
SmoothQuant ( <a href="#">Xiao et al., 2022</a> )	8/8/8	32	60.6	69.6	46.0	61.5	63.3	32.8	55.6
	MinMax FP Quant (E4M3)	32	78.0	79.1	60.0	72.3	77.2	47.1	<b>68.9</b>
	16/8/8	512	76.5	78.0	58.0	72.1	76.3	45.5	68.2
	FPQ baseline	32	78.0	79.1	59.9	72.3	77.2	47.1	<b>68.9</b>
	FPQ	32	78.1	78.5	59.1	72.4	76.4	46.1	68.4
SmoothQuant ( <a href="#">Xiao et al., 2022</a> )	4/4/8	32	52.1	65.0	36.4	53.9	52.3	29.0	48.1
	MinMax FP Quant (E2M1/E4M3)	32	78.0	78.9	58.0	71.6	76.0	44.8	<b>67.9</b>
	FPQ baseline	32	76.2	78.2	57.9	71.9	75.1	43.9	67.2
	FPQ	32	76.4	78.5	58.2	72.1	75.2	44.7	67.5
SmoothQuant ( <a href="#">Xiao et al., 2022</a> )	4/4/4	32	54.5	52.7	25.5	51.1	25.3	22.1	38.5
	MinMax FP Quant (E2M1)	32	45.8	51.7	25.5	49.5	25.0	22.8	36.7
	16/4/4	512	57.6	61.3	56.0	52.6	49.9	25.1	50.4
	FPQ baseline	32	54.3	57.7	35.7	52.2	41.1	25.7	44.5
	FPQ	32	71.9	74.8	53.3	66.7	71.7	39.9	<b>63.1</b>

Table 1: Zero-shot performance on common sense reasoning tasks with LLaMA ([Touvron et al., 2023](#)) models. We denote E/W/A as the bit-width of word embeddings, model weight and activations, respectively.

dataset following the setting of GPTQ ([Frantar et al., 2023](#)). The data preprocessing and score calculation are based on EleutherAI evaluation harness<sup>1</sup>. In Table 1, we compare FPQ to the floating-point PTQ baselines, and state-of-the-art PTQ and QAT methods, including SmoothQuant ([Xiao et al., 2022](#)) and GPTQ ([Frantar et al., 2023](#)), and LLM-QAT ([Liu et al., 2023](#)).

In general, all methods, except for the naïve Min-Max INT Quantization, produce comparable outcomes in the 8-bit setting on both LLaMA-7B and LLaMA-13B. Additionally, we observe that the naïve MinMax FP Quantization achieves nearly lossless results and even surpasses the state-of-the-art integer post-training quantization method, SmoothQuant ([Xiao et al., 2022](#)), which indicates that floating-point quantization naturally has a strong capability in handling the distributions in transformers. However, both MinMax FP Quant and FPQ baseline fail when pushing the quan-

tization precision to ultra-low 4/4/4 bit setting, with 28.9% and 23.8% accuracy degradation on LLaMA-7B, respectively. In this extreme case, the previous state-of-the-art PTQ and QAT methods, SmoothQuant ([Xiao et al., 2022](#)) and LLM-QAT ([Liu et al., 2023](#)) also suffer severe accuracy downgrade. In comparison, FPQ demonstrates a strong capability of handling extra-low bit settings and achieves only 8.2/5.8% accuracy drop on LLaMA-7B/13B with 4/4/4 bit-width, outperforming SmoothQuant ([Xiao et al., 2022](#)) by a large margin, yet with less bit-width and smaller calibration size. Moreover, FPQ even achieves 5.3% accuracy improvements compared to LLM-QAT ([Liu et al., 2023](#)) in the 4/4/4 setting and 1.5% over GPTQ ([Frantar et al., 2023](#)) in the 4/4/16 configuration on LLaMA-7B.

For practitioners, a crucial consideration is determining the appropriate quantization methods for various bit-widths. Therefore, based on our findings, we offer two recommendations that balance the trade-off between accuracy and

<sup>1</sup><https://github.com/EleutherAI/lm-evaluation-harness>

Quant Method	#Bits (E/W/A)	# Calib	MNLI <sub>-m</sub>	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	Avg.
(Full-precision)	32-32-32	-	84.9	91.4	92.1	93.2	59.7	90.1	86.3	72.2	83.7
MinMax INT Quant	8/8/8	128	77.0	89.9	88.9	92.9	51.8	88.2	83.8	71.5	80.5
MinMax FP Quant (E2M5)	8/8/8	128	78.9	90.8	88.6	92.9	52.7	88.4	84.3	69.0	80.7
MinMax FP Quant (E3M4)	8/8/8	128	84.5	90.9	91.5	93.2	58.3	<b>89.3</b>	87.7	71.8	83.4
MinMax FP Quant (E4M3)	8/8/8	128	<b>84.7</b>	90.9	<b>91.7</b>	93.0	58.6	<b>89.3</b>	86.5	<b>72.2</b>	83.4
MinMax FP Quant (E5M2)	8/8/8	128	84.1	90.9	91.4	<b>93.6</b>	58.1	89.2	87.5	71.8	83.3
FPQ baseline	8/8/8	128	84.6	90.9	<b>91.7</b>	93.1	58.6	<b>89.3</b>	<b>88.0</b>	<b>72.2</b>	83.5
FPQ	8/8/8	128	84.6	<b>91.0</b>	91.6	93.3	<b>58.8</b>	<b>89.3</b>	<b>88.0</b>	<b>72.2</b>	<b>83.6</b>
MinMax INT Quant	6/6/6	128	31.9	62.0	52.8	58.8	0.0	12.7	32.1	52.7	37.9
MinMax FP Quant (E2M3)	6/6/6	128	43.5	85.4	79.4	90.5	45.2	86.0	66.9	59.9	69.6
MinMax FP Quant (E3M2)	6/6/6	128	83.9	90.8	90.8	92.2	58.2	88.6	87.0	<b>72.2</b>	83.0
MinMax FP Quant (E4M1)	6/6/6	128	84.4	90.2	90.1	92.2	58.2	89.2	85.3	69.7	82.4
FPQ baseline	6/6/6	128	<b>84.6</b>	<b>90.9</b>	91.2	<b>93.2</b>	<b>58.8</b>	88.7	87.5	70.8	<b>83.2</b>
FPQ	6/6/6	128	84.5	90.8	<b>91.6</b>	93.1	57.3	<b>89.3</b>	<b>88.7</b>	71.8	<b>83.2</b>
MinMax INT Quant	4/4/8	128	33.1	63.8	60.1	49.3	0.0	44.0	50.2	49.1	43.7
MinMax FP Quant (E2M1)	4/4/8	128	60.6	70.9	77.4	79.9	5.5	78.6	46.8	56.6	59.5
MREM-S (Bai et al., 2022)	4/4/8	4096	83.5	90.2	<b>91.2</b>	91.4	55.1	89.1	84.8	71.8	82.1
MREM-P (Bai et al., 2022)	4/4/8	4096	83.4	90.2	91.0	91.5	54.7	89.1	86.3	71.1	82.2
FPQ baseline	4/4/8	128	84.4	<b>90.6</b>	<b>91.4</b>	<b>92.9</b>	58.6	83.7	88.2	<b>73.3</b>	82.9
FPQ	4/4/8	128	<b>84.5</b>	<b>90.6</b>	91.1	<b>92.7</b>	<b>58.8</b>	<b>89.3</b>	<b>88.7</b>	<b>73.3</b>	<b>83.6</b>
MinMax INT Quant	4/4/4	128	31.8	39.7	50.5	49.1	0.0	6.7	31.6	54.5	32.9
MinMax FP Quant (E2M1)	4/4/4	128	33.6	54.0	50.6	50.8	0.0	0.0	31.6	52.0	34.1
BrecQ (Li et al., 2021)	8/4/4	4096	31.9	62.3	50.7	50.9	0.9	6.4	31.7	52.3	35.8
QDrop (Wei et al., 2022)	8/4/4	4096	71.4	79.0	76.8	88.1	40.9	81.9	79.2	60.7	72.3
FPQ baseline	4/4/4	128	38.9	68.3	55.3	83.6	10.6	0.0	43.8	55.2	44.5
FPQ	4/4/4	128	<b>82.3</b>	<b>89.2</b>	<b>86.6</b>	<b>91.5</b>	<b>52.6</b>	<b>85.5</b>	<b>83.8</b>	<b>69.0</b>	<b>80.1</b>

Table 2: Results on the GLUE development set with BERT (Bai et al., 2022) model. We denote E/W/A as the bit-width of word embeddings, model weight and activations, respectively.

search/optimization efficiency. First of all, since the difference between MinMax FP Quant and the rest of the methods is marginal for the 8/8/8 setting, we recommend simply using the MinMax FP Quant method for the 8/8/8 setting as the MinMax method does not involve search process. However, for more demanding scenarios, especially with activation quantization to 4 bits, we recommend employing FPQ for minimizing accuracy degradation with negligible inference overhead.

## 5.2.2 BERT Model

We evaluate the proposed quantization techniques for BERT model on GLUE tasks (Wang et al., 2019). Full-precision BERT-base models fine-tuned on GLUE datasets are obtained from Huggingface public repository<sup>2</sup>. We randomly sample 128 data from the training set as the calibration set. In Table 2, FPQ demonstrates remarkable performance, achieving absolute average accuracy improvements of 44.3% compared to BrecQ (Li et al., 2021) and 7.9% over QDrop (Wei et al., 2022) with 4/4/4 bit setting. Further, with 4-bit weight and 8-bit activation, MREM-S/MREM-P (Bai et al., 2022) present a 1.6/1.5% accuracy gap to the full-precision model with 4096 calibration data, while FPQ achieves almost no accuracy loss with only

128 calibration data points.

## 5.2.3 Generalizability on Vision Transformer

Based on our findings that vision transformers also exhibit a consistent activation distribution pattern as language transformers, characterized by high inter-channel variance and low intra-channel variance, as detailed in Fig. 2, we extended our proposed methods to ViT and compared FPQ with floating-point PTQ baselines and state-of-the-art PTQ method for ViT on the ImageNet classification task. Table 3 shows that findings on ViT are consistent with that on language models: previous state-of-the-art integer-based methods struggled to maintain reasonable accuracy when quantizing the transformer to lower bits. In comparison, the proposed FPQ outperformed both PTQ4ViT and APQ-ViT on 6 bits, and also achieved 40.9% and 31.5% absolute accuracy improvement over PTQ4ViT and APQ-ViT on DeiT-S in the 4-bit configuration.

## 5.3 Ablation Study

In this section, we first compare the influence of different calibration sizes on FPQ. We vary the calibration size in {32, 64, 128, 256} and test on MNLI, QQP, and CoLA. Table 4 shows that the evaluation on MNLI and QQP is more robust to different settings, and the variance is more significant on CoLA. We observe that FPQ performs well with a

<sup>2</sup>[https://huggingface.co/textattack/bert-base-uncased-{TASK\\_NAME}](https://huggingface.co/textattack/bert-base-uncased-{TASK_NAME})

W/A	Quant Method	Deit-S	Deit-B	ViT-S
Full-prec	-	79.9	81.8	81.4
6/6	PTQ4ViT(Yuan et al., 2022)	76.3	80.3	78.6
6/6	APQ-ViT(Ding et al., 2022)	77.8	80.4	79.2
6/6	MinMax FP Quant (E3M2)	79.3	81.7	80.7
6/6	FPQ baseline	79.43	81.7	80.9
6/6	FPQ	<b>79.5</b>	<b>81.8</b>	<b>81.1</b>
4/4	PTQ4ViT(Yuan et al., 2022)	34.1	64.4	42.6
4/4	APQ-ViT (Ding et al., 2022)	43.6	67.5	48.0
4/4	MinMax FP Quant (E2M1)	0.4	0.1	0.1
4/4	FPQ baseline	6.57	0.71	0.3
4/4	FPQ	<b>75.0</b>	<b>79.4</b>	<b>73.2</b>

Table 3: Comparison on the ImageNet dataset with vision transformer structures.

E/W/A	#Calib	MNLI-M	QQP	CoLA
4/4/4	32	81.5	<b>89.4</b>	44.4
4/4/4	64	81.8	89.4	47.9
4/4/4	128	<b>82.3</b>	89.2	52.6
4/4/4	256	81.9	89.0	<b>52.9</b>
6/6/6	32	<b>84.8</b>	90.8	55.0
6/6/6	64	84.7	<b>90.9</b>	<b>58.2</b>
6/6/6	128	84.5	90.8	57.3
6/6/6	256	84.6	90.8	57.6

Table 4: Ablation studies of different calibration sizes.

calibration set size of 128 data points. However, we also find that it remains robust and maintains competitive accuracy even with limited access to calibration data, such as when using as few as 32 data points.

We investigate the robustness of FPQ to different search ranges ( $\gamma_1, \gamma_2$ ). Table 5 presents the results of FPQ using three sets of  $(\gamma_1, \gamma_2)$ : (0.01, 1.2), (0.1, 1.2), (0.5, 1.5), on MNLI, QQP, and CoLA. It is observed that no single search range outperforms the others consistently across all tasks. For instance, the search range (0.01, 1.2) performs better than (0.5, 1.5) on MNLI and QQP, but slightly worse on CoLA in the 4-bit configuration. Overall, FPQ exhibits robustness to various  $\gamma_1$  and  $\gamma_2$ , as long as the search range is not overly aggressive.

#### 5.4 Hardware Cost

We further examine the hardware utilization of low-bit INT, FP, and mixed-format FP multiplication operators, including adder, multiplier, and multiply-accumulate (MAC) units, in terms of hardware area. Mixed-format FP refers to the multiplication of floating-point numbers with different formats, e.g., E2M1 multiplies with E1M2. We implemented the MAC operator by Verilog HDL and utilized Cadence Genus to obtain the synthesized area under TSMC 40nm technology and 0.5GHz clock frequency.

Table 6 illustrates the hardware cost of the INT and FP operators, with the multiplier being the pri-

E/W/A	$\gamma_1, \gamma_2$	MNLI-M	QQP	CoLA
4/4/4	0.01, 1.2	<b>82.3</b>	<b>89.2</b>	52.6
4/4/4	0.1, 1.2	82.2	89.1	<b>53.6</b>
4/4/4	0.5, 1.5	82.3	88.4	52.8
6/6/6	0.01, 1.2	84.5	90.8	57.3
6/6/6	0.1, 1.2	<b>84.7</b>	90.8	57.5
6/6/6	0.5, 1.5	84.7	<b>90.8</b>	<b>57.8</b>

Table 5: Ablation studies of different search range.

Format	Adder( $\mu m^2$ )	Multiplier( $\mu m^2$ )	MAC( $\mu m^2$ )
INT4	93	182	410
INT6	132	340	529
E2M1	111	92	443
E3M2	223	138	498
E2M1 * E1M2	105	107	432

Table 6: Area differences of INT, FP and mixed Format FP operators across different bit-widths.

mary cost for INT and the adder for FP. Notably, the disparity between FP4 and INT4 adders is small, while INT has twice the hardware cost for the multiplier. Moreover, the mixed-format FP4 operator has comparable hardware area as the standard FP4 operator. These findings indicate that the proposed FPQ approach imposes negligible overhead in terms of hardware implementation when compared to the standard FP operators and the hardware cost for FP is comparable with INT.

## 6 Conclusion

This paper presents the first successful demonstration of 4-bit floating-point post-training quantization for weights, activations, and embeddings in natural language transformer architectures, including both large language models and BERT model. We also extend our method to vision transformers and observe its robust generalization ability. Our approach involves a practical search-based technique which establishes a strong baseline and achieves state-of-the-art results for 6-bit and 8-bit quantization. Furthermore, we address the challenge of high inter-channel variance in transformers by proposing *pre-shifted exponent bias*, which proves highly effective in achieving accurate 4-bit quantization.

## Acknowledgement

This research is supported by National Natural Science Foundation of China/ HKSAR Research Grants Council Joint Research Scheme under Grant *NHKUST627/20*, and Foshan HKUST Projects under Grant *FSUST21 – HKUST10E*.

## Limitations

Our experiments were conducted on publicly available datasets with finite sentence lengths, and the generalizability of our method to extremely long sequences or streaming data has not been verified and may require further investigation. In addition, it remains to be seen how our proposed method can generalize to other domains beyond language and vision, such as audio. It would also be interesting to see the applicability of our method to generative tasks and other applications.

## References

- Hassan Akbari, Liangzhe Yuan, Rui Qian, Wei-Hong Chuang, Shih-Fu Chang, Yin Cui, and Boqing Gong. 2021. Vatt: Transformers for multimodal self-supervised learning from raw video, audio and text. *Advances in Neural Information Processing Systems*, 34:24206–24221.
- Haoli Bai, Lu Hou, Lifeng Shang, Xin Jiang, Irwin King, and Michael Lyu. 2022. Towards efficient post-training quantization of pre-trained language models. In *Advances in Neural Information Processing Systems*.
- Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. 2021. Understanding and overcoming the challenges of efficient transformer quantization.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.
- Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2020. Zeroq: A novel zero shot quantization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13169–13178.
- Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. 2019. Low-bit quantization of neural networks for efficient inference.
- Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35:30318–30332.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding.
- Yifu Ding, Haotong Qin, Qinghua Yan, Zhenhua Chai, Junjie Liu, Xiaolin Wei, and Xianglong Liu. 2022. Towards accurate post-training quantization for vision transformer. In *Proceedings of the 30th ACM International Conference on Multimedia, MM ’22*, page 5380–5388, New York, NY, USA. Association for Computing Machinery.
- Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*.
- Elias Frantar, Saleh Ashkboos, Torsten Hoefer, and Dan Alistarh. 2023. GPTQ: Accurate post-training compression for generative pretrained transformers. In *International Conference on Learning Representations*.
- Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.
- Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186.
- Andrey Kuzmin, Mart Van Baalen, Yuwei Ren, Markus Nagel, Jorn Peters, and Tijmen Blankevoort. 2022. Fp8 quantization: The power of the exponent. *Advances in Neural Information Processing Systems*, 35:14651–14662.
- Jemin Lee, Yongin Kwon, Jeman Park, Misun Yu, and Hwanjun Song. 2023. Q-hyvit: Post-training quantization for hybrid vision transformer with bridge block reconstruction.
- Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880.
- Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. 2021. Brecq: Pushing the limit of post-training quantization by block reconstruction. *arXiv preprint arXiv:2102.05426*.
- Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. 2023. Llm-qat: Data-free quantization aware training for large language models. *arXiv preprint arXiv:2305.17888*.
- Zhenhua Liu, Yunhe Wang, Kai Han, Wei Zhang, Siwei Ma, and Wen Gao. 2021. Post-training quantization for vision transformer. *Advances in Neural Information Processing Systems*, 34:28092–28103.

- Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. 2022. [Fp8 formats for deep learning](#).
- Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. 2020. Up or down? adaptive rounding for post-training quantization. In *International Conference on Machine Learning*, pages 7197–7206. PMLR.
- Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. 2019. [Data-free quantization through weight equalization and bias correction](#).
- OpenAI. 2023. Gpt-4 technical report. *ArXiv*, abs/2303.08774.
- Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551.
- Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. 2021. Training data-efficient image transformers & distillation through attention. In *International conference on machine learning*, pages 10347–10357. PMLR.
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.
- Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. [Glue: A multi-task benchmark and analysis platform for natural language understanding](#).
- Xiuying Wei, Ruihao Gong, Yuhang Li, Xianglong Liu, and Fengwei Yu. 2022. [QDrop: Randomly dropping quantization for extremely low-bit post-training quantization](#). In *International Conference on Learning Representations*.
- Di Wu, Qi Tang, Yongle Zhao, Ming Zhang, Ying Fu, and Debng Zhang. 2020. [Easyquant: Post-training quantization via scale optimization](#).
- Guangxuan Xiao, Ji Lin, Mickael Seznec, Julien Demouth, and Song Han. 2022. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*.
- Zhihang Yuan, Chenhao Xue, Yiqi Chen, Qiang Wu, and Guangyu Sun. 2022. Ptq4vit: Post-training quantization for vision transformers with twin uniform quantization. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XII*, pages 191–207. Springer.
- Yijia Zhang, Lingran Zhao, Shijie Cao, Wenqiang Wang, Ting Cao, Fan Yang, Mao Yang, Shanghang Zhang, and Ningyi Xu. 2023. [Integer or floating point? new outlooks for low-bit quantization on large language models](#).

## A Hessian-Based Loss Metric

The objective of post-training quantization is to minimize the perturbation ( $\delta\mathbf{X} = \mathbf{X}_{\text{FP}} - \mathbf{X}_R$ ) introduced by quantization to the pre-trained real-valued network:

$$\min \mathbb{E}[\mathcal{L}(\mathbf{X}_R + \delta\mathbf{X}) - \mathcal{L}(\mathbf{X}_R)] \quad (20)$$

Following the Taylor series expansion, we have

$$\begin{aligned} & \mathbb{E}[\mathcal{L}(\mathbf{X}_R + \delta\mathbf{X}) - \mathcal{L}(\mathbf{X}_R)] \\ & \approx \delta\mathbf{X}^T \bar{\mathbf{g}}^{(\mathbf{X})} + \frac{1}{2} \delta\mathbf{X}^T \bar{\mathbf{H}}^{(\mathbf{X})} \delta\mathbf{X} \\ & \approx \frac{1}{2} \delta\mathbf{X}^T \bar{\mathbf{H}}^{(\mathbf{X})} \delta\mathbf{X} \end{aligned} \quad (21)$$

Here,  $\bar{\mathbf{g}}^{(\mathbf{X})}$  is the gradients and  $\bar{\mathbf{H}}^{(\mathbf{X})}$  is the Hessian matrix. Since the pre-trained model is well-converged, we can assume that  $\bar{\mathbf{g}}^{(\mathbf{X})}$  has near zero value in every element, and thus term  $\delta\mathbf{X}^T \bar{\mathbf{g}}^{(\mathbf{X})}$  can be neglected.

The Hessian matrix  $\bar{\mathbf{H}}^{(\mathbf{X})}$  is computed as:

$$\bar{\mathbf{H}}^{(\mathbf{X})} = \mathbf{J}_O^T(\mathbf{X}) \bar{\mathbf{H}}^{(O)} \mathbf{J}_O(\mathbf{X}) \quad (22)$$

where  $\mathbf{J}_O(\mathbf{X})$  denotes the Jacobian matrix of the layer output  $O$  w.r.t  $\mathbf{X}$ , and  $\bar{\mathbf{H}}^{(O)}$  is the Hessian matrix w.r.t  $O$ . We then substitute the above equation back to equation 21 :

$$\begin{aligned} & \delta\mathbf{X}^T \bar{\mathbf{H}}^{(\mathbf{X})} \delta\mathbf{X} \\ & = (\mathbf{J}_O(\mathbf{X}) \delta\mathbf{X})^T \bar{\mathbf{H}}^{(O)} (\mathbf{J}_O(\mathbf{X}) \delta\mathbf{X}) \\ & \approx (\hat{\mathbf{O}} - \mathbf{O})^T \bar{\mathbf{H}}^{(O)} (\hat{\mathbf{O}} - \mathbf{O}) \end{aligned} \quad (23)$$

Here  $\hat{\mathbf{O}}$  is the intermediate output of the quantized layer and  $\mathbf{O}$  is the original layer output. Note that under the assumption that  $\delta\mathbf{X}$  is relatively small (Li et al., 2021), we can approximate  $(\hat{\mathbf{O}} - \mathbf{O})$  as  $\mathbf{J}_O(\mathbf{X}) \delta\mathbf{X}$  using first-order Taylor expansion.

Nevertheless, the calculation of  $\bar{\mathbf{H}}^{(\hat{\mathbf{O}})}$  is still burdensome, therefore, we use the diagonal entries of the Fisher Information Matrix of  $\mathbf{O}$  to substitute  $\bar{\mathbf{H}}^{(O)}$  following (Li et al., 2021; Yuan et al., 2022), and the new Hessian-based metric becomes:

$$\mathbb{E}[(\hat{\mathbf{O}} - \mathbf{O})^T \text{diag}((\frac{\partial L}{\partial \mathbf{O}_1})^2, \dots, (\frac{\partial L}{\partial \mathbf{O}_n})^2) (\hat{\mathbf{O}} - \mathbf{O})] \quad (24)$$

Here, each entry of  $\mathbf{O}$  is assumed to be independent and  $n$  denoted the total number of elements in  $\mathbf{O}$ . In this study, this hessian-based metric is used as the reconstruction metric to search for the optimal FP quantization function for both the weight and activation when performing layer-wise reconstruction in BERT and Vision Transformer models.

## B Quantization Error of Different Floating-Point Formats

Figure 4 compares the quantization error of different formats in 8-bit quantization, including INT8, E2M5, E3M4, E4M3, and E5M2. We apply these formats to different BERT modules in the first, fifth, and last layers. The figures demonstrate that the optimal FP formats differs depending on the specific module that we are quantizing.

## C Inter-Channel Variance Visualization

Figure 5 and 6 depict the output of different fully-connected layers in BERT for the MNLI task, DeiT-S for the ImageNet-1K task, and LLaMA-7B for the zero-shot reasoning task. The visualizations reveal a noticeable inter-channel variance presented in both language and vision transformers.

## D Efficient Matrix Multiplication

Figure 7 displays a comprehensive list of all the granularity options that allow for efficient matrix multiplication. While per-token quantization theoretically provides greater precision in terms of quantization granularity, the accuracy gains achieved through this method are minimal and do not justify the additional computational overhead required. As a result, we have opted to use per-tensor quantization when quantizing activations.

## E Learning Format and Maximum Value

We compare the previous gradient-based method (Kuzmin et al., 2022) with the proposed search-based method for finding the optimal format and maximum value. On DeiT-S, the learnable method only achieves 74.38% accuracy for an 8-bit quantized model on ImageNet, in contrast, FPQ can attain an almost loss-less result of 79.88%. We analyze the gradients for the number of exponent bits  $e$  derived in (Kuzmin et al., 2022) and observe that each time the exponent bits change, the gradients experience exponential variations, leading to high instability. Based on this observation, we assert that employing a search-based method to determine the optimal formats is crucial in post-training quantization (PTQ).

## F Reconstruction Choices

The previous works on integer post-training quantization involves breaking down the target model into

sub-modules and reconstructing them separately (Nagel et al., 2020; Li et al., 2021; Bai et al., 2022; Yuan et al., 2022). This addresses the problem of over-fitting, given that only a limited amount of unlabeled calibration data is available. In this study we find the layer-wise reconstruction and parallel quantization works best for floating-point PTQ:

**Layer Reconstruction:** Recent research (Li et al., 2021; Bai et al., 2022) suggests increasing the reconstruction granularity from layer reconstruction (Nagel et al., 2020) to block reconstruction (Li et al., 2021) or even larger granularity (Lee et al., 2023). This is achieved by jointly optimizing all the linear layers or matrix multiplication components within each module to prevent the propagation of reconstruction errors among the layers. Despite this, we have observed that increasing the recon-

struction granularity does not improve the accuracy of FPQ baseline or sometimes even lead to worse results. Therefore, we choose layer reconstruction.

**Parallel Quantization:** Sequential quantization is the most commonly used approach (Wu et al., 2020; Nagel et al., 2020; Li et al., 2021) where modules are quantized consecutively based on their sequential order, and the input for the current calibrating module is generated using all the previously quantized modules. However, some recent works (Yuan et al., 2022; Bai et al., 2022) proposed a new parallel quantization framework. This framework uses the raw output of the full-precision modules as input and makes the calibration of each module independent from one another. In this work, we use parallel quantization, as it yields better results than its sequential counterparts.

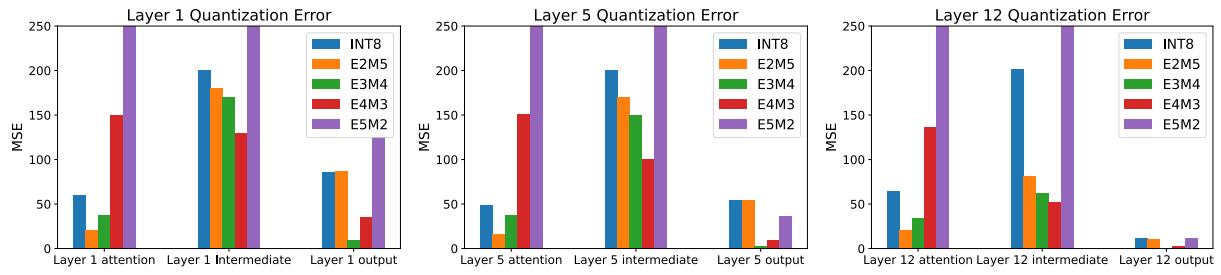


Figure 4: Quantization error of different formats for BERT layers.

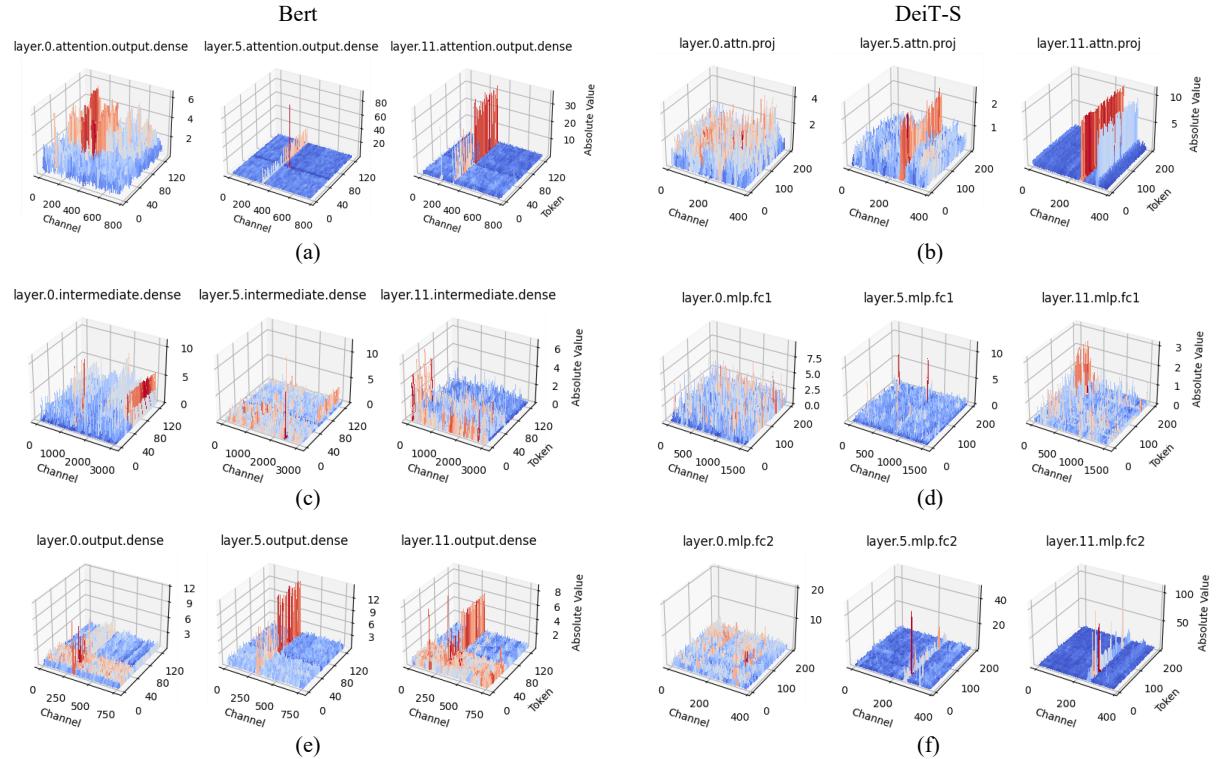


Figure 5: Magnitude of the output activations of different modules in BERT (left column), and DeiT-S (right column).

### LLaMa-7B

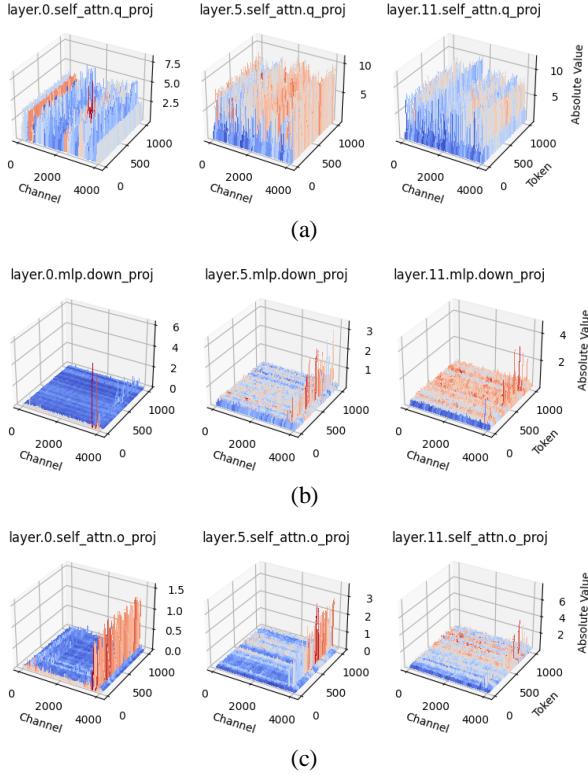


Figure 6: Magnitude of the output activations of different modules in LLaMA-7B.

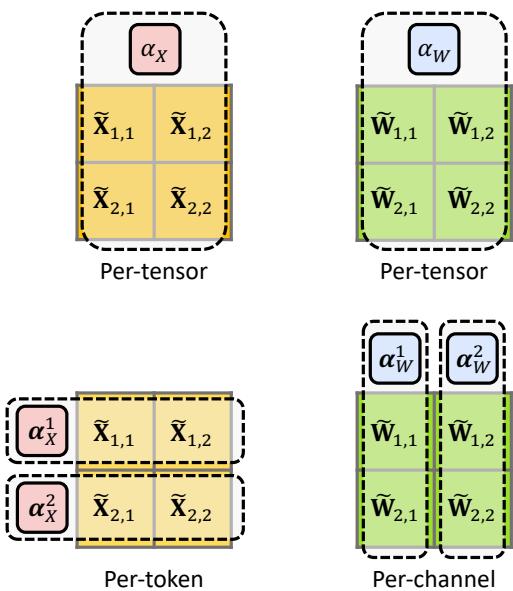


Figure 7: Quantization granularity options that support efficient matrix multiplication. The dimensions that share the same scaling factor are indicated with red dotted frames

# SHEARED LLaMA: ACCELERATING LANGUAGE MODEL PRE-TRAINING VIA STRUCTURED PRUNING

**Mengzhou Xia<sup>1</sup>, Tianyu Gao<sup>1</sup>, Zhiyuan Zeng<sup>2\*</sup>, Danqi Chen<sup>1</sup>**

<sup>1</sup>Department of Computer Science & Princeton Language and Intelligence,  
Princeton University

<sup>2</sup>Department of Computer Science and Technology, Tsinghua University  
 {mengzhou, tianyug, danqic}@cs.princeton.edu  
 zengzy20@mails.tsinghua.edu.cn

## ABSTRACT

The popularity of LLaMA (Touvron et al., 2023a;b) and other recently emerged moderate-sized large language models (LLMs) highlights the potential of building smaller yet powerful LLMs. Regardless, the cost of training such models from scratch on trillions of tokens remains high. In this work, we study structured pruning as an effective means to develop smaller LLMs from pre-trained, larger models. Our approach employs two key techniques: (1) *targeted structured pruning*, which prunes a larger model to a specified target shape by removing layers, heads, and intermediate and hidden dimensions in an end-to-end manner, and (2) *dynamic batch loading*, which dynamically updates the composition of sampled data in each training batch based on varying losses across different domains. We demonstrate the efficacy of our approach by presenting the **Sheared-LLaMA** series, pruning the LLaMA2-7B model down to 1.3B and 2.7B parameters. Sheared-LLaMA models outperform state-of-the-art open-source models of equivalent sizes, such as Pythia, INCITE, and OpenLLaMA models, on a wide range of downstream and instruction tuning evaluations, while requiring only 3% of compute compared to training such models from scratch. This work provides compelling evidence that leveraging existing LLMs with structured pruning is a far more cost-effective approach for building smaller LLMs.<sup>1</sup>

## 1 INTRODUCTION

Large language models (LLMs) are extremely performant on a wide range of natural language tasks, but they require enormous amounts of compute to train (OpenAI, 2023; Anthropic, 2023). As such, there is growing interest in building strong moderate-sized models, such as LLaMA (Touvron et al., 2023a;b), MPT (MosaicML, 2023), and Falcon (Almazrouei et al., 2023), that allow for efficient inference and fine-tuning. These LLMs are available in varied sizes suited for different use cases, but training each individual model from scratch—even the smallest billion-parameter models—requires substantial computational resources that are cost-prohibitive for most organizations. In this work, we seek to address the following question:

*Can we produce a smaller, general-purpose, and competitive LLM by leveraging existing pre-trained LLMs, while using much less compute than training one from scratch?*

We explore structured pruning as a means to achieve this goal. Pruning is commonly viewed as a solution for compressing task-specific models (Han et al., 2016; Li et al., 2016; Lagunas et al., 2021; Xia et al., 2022; Kurtic et al., 2023), removing redundant parameters and accelerating inference without sacrificing task performance. However, for general-purpose LLMs, pruning inevitably results in performance degradation compared to original models (Frantar & Alistarh, 2023; Sun et al., 2023; Ma et al., 2023), especially when without significant compute invested post-pruning. In this

\*Work done during internship at Princeton University.

<sup>1</sup>Please find our code and models at <https://github.com/princeton-nlp/LLM-Shearing>.

work, we use pruning as an effective approach for developing smaller yet competitive LLMs that require only a fraction of the training compute compared to training them from scratch.

We identify two key technical challenges in this problem. First, how can we decide on final pruned architectures that are strong in performance and efficient for inference? Existing structured pruning techniques for LLMs (Xia et al., 2022; Ma et al., 2023) do not specify targeted structures and lead to suboptimal pruned models in terms of performance and inference speed (Table 4 and Figure 8). Second, how can we continue pre-training the pruned model to reach desired performance? We observe that training using the original pre-training data leads to imbalanced rates of loss reduction across different domains, compared to when training such models from scratch. This indicates that the pruned model retains varying levels of knowledge for different domains (e.g., GitHub vs. C4) and simply using the pre-training domain proportion results in an inefficient use of data (Figure 5). To address these issues, we propose “LLM-shearing”, an algorithm consisting of the following two components:

- We propose a novel pruning algorithm, dubbed *targeted structured pruning*, which prunes a source model to a specified target architecture. The target architecture is determined by leveraging the configurations of existing pre-trained models. Our pruning approach searches for substructures within the source model that maximally preserve performance while adhering to the given constraints.
- We devise a *dynamic batch loading* algorithm that loads training data from each domain in proportion to its rate of loss reduction, thereby making an efficient use of the data and accelerating the overall performance improvement.

We demonstrate the efficacy of our proposed method by pruning a LLaMA2-7B model (Touvron et al., 2023b) into two smaller LLMs: Sheared-LLaMA-1.3B and Sheared-LLaMA-2.7B. Despite using only 50 billion tokens (i.e., 5% of OpenLLaMA’s pre-training budget) for pruning and continued pre-training, Sheared-LLaMA-1.3B and Sheared-LLaMA-2.7B outperform other popular LLMs at similar scales, including Pythia (Biderman et al., 2023), INCITE (TogetherAI, 2023b), and OpenLLaMA (Geng & Liu, 2023), on 11 representative downstream tasks (Figure 1; commonsense, reading comprehension, and world knowledge) and instruction tuning for open-ended generation. Additionally, the downstream performance trajectory suggests that further training the pruned model with more tokens would result in even greater gains. While we only conduct experiments with up to 7B parameter models, our LLM-shearing algorithm is highly generalizable and can be extended to large language models of any size in future work.

## 2 LLM-SHEARING

Given an existing large model  $\mathcal{M}_S$  (the *source* model), we study how to efficiently produce a smaller, strong model  $\mathcal{M}_T$  (the *target* model). We consider this as a two-stage process: (1) Pruning  $\mathcal{M}_S$  into  $\mathcal{M}_T$ . This reduces the number of parameters but incurs a performance drop inevitably. (2) Continually pre-training  $\mathcal{M}_T$  with a standard language modeling objective to reach a target performance. While most recent efforts (Xia et al., 2022; Ma et al., 2023) focus on the former stage, we find the latter stage crucial for producing competitive general-purpose LLMs from structured pruning.

### 2.1 TARGETED STRUCTURED PRUNING

Structured pruning removes groups of model parameters to compress models and accelerate inference. However, existing structured pruning approaches often result in unconventional model configurations that deviate from popular architectures. For example, CoFiPruning (Xia et al., 2022) pro-

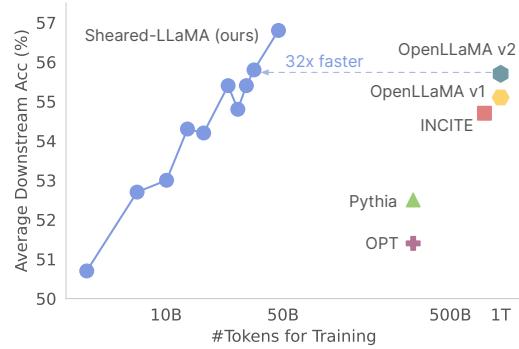


Figure 1: Our Sheared-LLaMA-2.7B surpasses a series of open-source models at a similar scale and only requires 1/32 (3%) of budget to achieve on-par performance with OpenLLaMA-3B-v2.

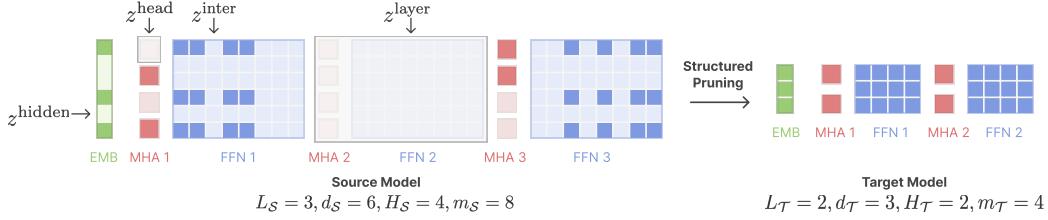


Figure 2: *Targeted structured pruning* produces a compact and dense model of a pre-specified shape. Light colors indicate pruned substructures. Masking variables  $z$  are learned to control whether a substructure is pruned ( $z = 0$ ) or retained ( $z = 1$ ).

duces models with non-uniform layer configurations (e.g., different numbers of heads across layers), which incurs inference overhead compared to standard uniform layer configurations (Section 4.2).

In this work, we extend CoFiPruning to allow pruning the source model into any target configuration that we specify. We leverage the configurations of existing pre-trained models as the target architectures, based on the intuition that these configurations have already been well-optimized to balance model expressivity and inference efficiency. For example, we use the INCITE-Base-3B architecture (TogetherAI, 2023a) as the target structure when producing a 2.7B model.

Our method learns a set of pruning masks on model parameters at different granularities—from global ones like layers and hidden dimensions (persist across all layers), to local ones like attention heads and intermediate dimensions. Assume that the source model  $\mathcal{M}_S$  has  $L_S$  layers, with each layer consisting of one multi-head attention module (MHA) and one feed-forward network (FFN).  $\mathcal{M}_S$  has a hidden state dimension of  $d_S$ ,  $H_S$  heads in each MHA, and an intermediate dimension of  $m_S$ . We introduce the following mask variables:

Granularity	Layer	Hidden dimension	Head	Intermediate dimension
Pruning masks	$z^{\text{layer}} \in \mathbb{R}^{L_S}$	$z^{\text{hidden}} \in \mathbb{R}^{d_S}$	$z^{\text{head}} \in \mathbb{R}^{H_S} (\times L_S)$	$z^{\text{int}} \in \mathbb{R}^{m_S} (\times L_S)$

Each mask variable controls whether the associated substructure is pruned or retained. For example, we remove a layer if its corresponding  $z^{\text{layer}} = 0$ . Figure 2 illustrates an example of how the pruning masks control the pruned structures.

We formulate pruning as a constrained optimization problem where we learn pruning masks to search for a subnetwork matching a pre-specified target architecture while maximizing performance. Following the  $\ell_0$  regularization approach (Louizos et al., 2018), we parametrize the pruning masks to model hard concrete distributions. These distributions have support on  $[0, 1]$  but concentrate their probability mass at 0 or 1, enabling discrete prune or retain decisions. While prior work usually control for a target sparsity (Wang et al., 2020; Xia et al., 2022), we use a pair of Lagrange multipliers to impose constraints on the pruned model shape directly. For example, for a target number of heads  $H_T$  (and we use  $L_T$ ,  $d_T$ , and  $m_T$  to represent the target number of layers, hidden dimension, and intermediate dimension respectively), we have the imposed constraint on a single layer as:

$$\tilde{\mathcal{L}}^{\text{head}}(\lambda, \phi, z) = \lambda^{\text{head}} \cdot \left( \sum z^{\text{head}} - H_T \right) + \phi^{\text{head}} \cdot \left( \sum z^{\text{head}} - H_T \right)^2.$$

Similar constraints are applied to pruning other substructures. Overall, we jointly optimize the model weights and pruning masks by a min-max objective  $\min_{\theta, z} \max_{\lambda, \phi} \mathcal{L}_{\text{prune}}(\theta, z, \lambda, \phi)$ :

$$\mathcal{L}_{\text{prune}}(\theta, z, \lambda, \phi) = \mathcal{L}(\theta, z) + \sum_{j=1}^{L_S} \tilde{\mathcal{L}}_j^{\text{head}} + \sum_{j=1}^{L_S} \tilde{\mathcal{L}}_j^{\text{int}} + \tilde{\mathcal{L}}^{\text{layer}} + \tilde{\mathcal{L}}^{\text{hidden}},$$

where  $\mathcal{L}(\theta, z)$  is the language modeling loss computed with the masked model weights. This objective will produce a pruned model with the target shape. Ideally, running this prune algorithm on a large amount of data will directly produce a strong compact model. In practice, the pruning stage is expensive (roughly 5× slower compared to standard LM training), and we find that the learned masks often converge fast. Therefore, in our experiments, we allocate only a limited budget

**Algorithm 1:** Dynamic Batch Loading

---

**Require:** Training data of  $k$  domains  $D_1, D_2, \dots, D_k$ , validation data  $D_1^{\text{val}}, D_2^{\text{val}}, \dots, D_k^{\text{val}}$ , initial data loading weights  $w_0 \in \mathbb{R}^k$ , reference loss  $\ell_{\text{ref}} \in \mathbb{R}^k$ , LM loss function  $\mathcal{L}$  or pruning loss  $\mathcal{L}_{\text{prune}}$ , training steps  $T$ , evaluation interval  $m$ , model parameters  $\theta (\theta, z, \phi, \lambda$  for pruning)

```

for  $t = 1, \dots, T$  do
    if  $t \bmod m = 0$  then
         $\ell_t[i] \leftarrow \mathcal{L}(\theta, z, D_i^{\text{val}})$  if pruning else  $\mathcal{L}(\theta, D_i^{\text{val}})$ 
         $\Delta_t[i] \leftarrow \max \{\ell_t[i] - \ell_{\text{ref}}[i], 0\}$   $\triangleright$  Calculate loss difference
         $w_t \leftarrow \text{UpdateWeight}(w_{t-m}, \Delta_t)$   $\triangleright$  Update data loading proportion
    end
    Sample a batch of data  $\mathcal{B}$  from  $D_1, D_2, \dots, D_k$  with proportion  $w_t$ ;
    if pruning then
        Update  $\theta, z, \phi, \lambda$  with  $\mathcal{L}_{\text{prune}}(\theta, z, \phi, \lambda)$  on  $\mathcal{B}$ 
    else
        Update  $\theta$  with  $\mathcal{L}(\theta, \mathcal{B})$ 
    end
end

Subroutine UpdateWeight ( $w, \Delta$ )
     $\alpha \leftarrow w \cdot \exp(\Delta)$   $\triangleright$  Calculate the unnormalized weights
     $w \leftarrow \frac{\alpha}{\sum_i \alpha[i]}$  return  $w$   $\triangleright$  Renormalize the data loading proportion
return  $\theta$ 

```

---

for the pruning process. Following pruning, we finalize the pruned architecture by preserving the highest-scoring components associated with the mask variables in each substructure, and continue pre-training the pruned model with the language modeling objective. We refer to this second stage as continued pre-training.

## 2.2 DYNAMIC BATCH LOADING

Continued pre-training on a large amount of data is crucial for recovering the pruned model performance. However, we observe a surprising finding in our preliminary experiments: continuing pre-training our pruned models on an existing pre-training dataset RedPajama (TogetherAI, 2023b; LLaMA’s pre-training dataset) reduces loss at different rates across domains compared to pre-training a model from scratch, which signifies an inefficient use of data.

To be more specific, we first fit a *scaling law* (Hoffmann et al., 2022; details in Appendix A) on the series of LLaMA2 models for each domain. Then we predict the loss that a hypothetical 2.7B LLaMA2 model would achieve if trained from scratch on the same data. We obtain these estimated *reference losses* across domains of the pre-training data and compare them to the losses of our pruned model after continued pre-training. As shown in Figure 5 (left), while our model’s loss on GitHub is better than the reference loss, it is significantly worse than the reference loss on C4. This observation indicates that pruning preserves a greater amount of knowledge in low-entropy and smaller domains (e.g., GitHub) compared to high-entropy and larger domains (e.g., C4). As demonstrated later in Section 4.1, simply reusing the original pre-training data distribution<sup>2</sup> results in an inefficient use of data and worse downstream performance, even if the overall loss is seemingly low.

Inspired by recent work (Xie et al., 2023), we propose *dynamic batch loading*, a more efficient algorithm to simply adjust domain proportions on the fly based on the model performance. The goal is to ensure the model achieves the reference loss roughly simultaneously across all domains. We introduce the algorithm below.

**Problem setup.** The pre-training data comprises of  $k$  domains  $D_1, D_2, \dots, D_k$  and we have a held-out validation dataset for each domain, denoted as  $D_i^{\text{val}}$ . At each training step  $t$ , a proportion  $w_t[i]$  of the data comes from domain  $D_i$ . We set a reference validation loss  $\ell_{\text{ref}}(D_i)$  for each domain and train the pruned model to reach the reference loss.

---

<sup>2</sup>The LLaMA2 pre-training data is not public. We conducted the same analysis on LLaMA1 models and observed a similar phenomenon, indicating that this is a universal issue unrelated to specific pre-training data.

**Dynamic batch loading.** We present the full algorithm in Algorithm 1. In a sketch, for every  $m$  steps, we evaluate the model to get the validation loss  $\ell_t$  (step  $t$ ) on  $D^{\text{val}}$ , and update  $w_t$  based on the difference  $\Delta_t(D_i)$  between  $\ell_{\text{ref}}[i]$  and  $\ell_t[i]$  on each domain. The update rule is exponential ascent following Xie et al. (2023),

$$\alpha_t = w_{t-m} \cdot \exp(\Delta_t); \quad w_t = \frac{\alpha_t}{\sum_i \alpha_t[i]}.$$

We apply dynamic batch loading to both the pruning stage and the continued pre-training stage. For pruning, we use the original pre-training data’s domain weights as  $w_0$ . For continued pre-training, we use the final weights from the pruning stage as  $w_0$ . Dynamic batch loading leverages reference losses on validation sets and adjusts the weights dynamically, so it adds minimal overhead to standard training. This improves the efficiency of Xie et al. (2023), which requires training both a reference and a proxy model to learn domain weights before training.

More broadly, dynamic batch loading has the potential to train an LLM to match reference losses of any model, by leveraging open-source pre-training datasets such as RedPajama, even when the reference model’s training data is unknown.

**Choices of reference losses.** By default, we use the loss predicted by the scaling law as the reference (denoted as *scaling reference*). We also experiment with an alternative where we directly use the source model’s domain validation loss as the reference (denoted as *source reference*). We show in Appendix E.3 and E.4 that while both variants perform well, using scaling reference leads to slightly better downstream results, especially on math and coding tasks. However, source reference is a viable alternative when a series of source models at different scales is not available.

### 3 EXPERIMENTS

#### 3.1 SETUP

**Model configurations.** We use the LLaMA2-7B model (Touvron et al., 2023b) as the source model throughout all of our main experiments.<sup>3</sup> We then conduct structured pruning experiments to compress this model down to two smaller target sizes—2.7B and 1.3B parameters. We compare to strong pre-trained language models of similar sizes, including OPT-1.3B (Zhang et al., 2022), Pythia-1.4B (Biderman et al., 2023), OPT-2.7B, Pythia-2.8B, INCITE-Base-3B (TogetherAI, 2023b), OpenLLaMA-3B-v1, and OpenLLaMA-3B-v2 (Geng & Liu, 2023). We use Pythia-1.4B as the target architecture for the 1.3B model, and INCITE-Base-3B as the target architecture for the 2.7B model. Table 8 summarizes model architecture details of all these models.

**Data.** As the training data for LLaMA2 is not publicly accessible, we use RedPajama (TogetherAI, 2023b), which is a replicated pre-training dataset of the LLaMA1 models (Touvron et al., 2023a), for pruning and continued-pretraining. This dataset encompasses training data from seven domains: CommonCrawl, C4, Github, Wikipedia, Books, ArXiv, and Stack-Exchange. We construct a held-out validation set with 2 million tokens (equivalent to 500 sequences of 4,096 tokens) for each domain. We allocate 0.4 billion tokens for the pruning phase and 50 billion tokens for the continued pre-training process. Following the conventions of LLaMA2, we maintain a sequence length of 4,096 tokens. Table 1 provides a summary of the pre-training data used by our models and the baseline models.

Table 1: A summary of pre-training datasets used by Sheared-LLaMA and other models.

Model	Pre-training Data	#Tokens
LLaMA1	LLaMA data	1T
LLaMA2	Unknown	2T
OPT	OPT data <sup>4</sup>	300B
Pythia	The Pile	300B
INCITE-Base	RedPajama	800B
OpenLLaMA v1	RedPajama	1T
OpenLLaMA v2	OpenLLaMA data <sup>5</sup>	1T
Sheared-LLaMA	RedPajama	50B

<sup>3</sup>Please find results on LLaMA1 models in Appendix E.5.

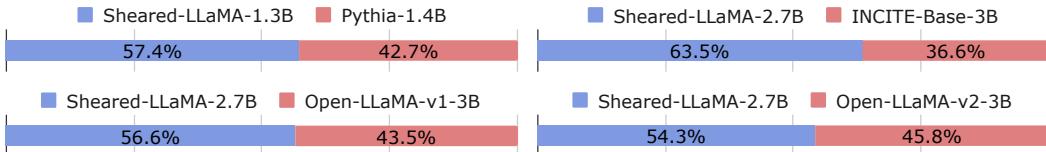


Figure 3: Sheared-LLaMAs outperform Pythia-1.4B, INCITE-Base-3B, OpenLLaMA-3B-v1 and OpenLLaMA-3B-v2 in instruction tuning.

**Training.** Our implementation builds on the `Composer` package (MosaicML, 2021). We use a maximum of 16 Nvidia A100 GPUs (80GB) for all experiments (More details are in Appendix B).

**Downstream task evaluation.** We use the `lm-evaluation-harness` package (Gao et al., 2021) to evaluate on an extensive suite of downstream tasks:

- We follow Pythia and LLaMA2 to report the 0-shot accuracy of ARC easy (ARC-E; Clark et al., 2018), LAMBADA (Paperno et al., 2016), LogiQA (Liu et al., 2020), PIQA (Bisk et al., 2020), SciQ (Welbl et al., 2017), and WinoGrande (Sakaguchi et al., 2021).
- We report accuracy of the tasks used by Open LLM Leaderboard<sup>6</sup>, including 10-shot Hellaswag (Zellers et al., 2019), 25-shot ARC Challenge (ARC-C; Clark et al., 2018), and 5-shot MMLU (Hendrycks et al., 2021).
- We also report exact match of 32-shot Natural Questions (NQ; Kwiatkowski et al., 2019) to measure the factual knowledge in the model.

**Instruction tuning evaluation.** As training models to follow instructions has become a crucial application of LLMs (Ouyang et al., 2022; Taori et al., 2023), we evaluate our models on instruction tuning and fine-tune both Sheared-LLaMA and baseline models on 10,000 instruction-response pairs sampled from the ShareGPT dataset<sup>7</sup>. For evaluation, we sample another 1,000 instructions from ShareGPT, generate responses from our fine-tuned models and other baseline models, and use GPT-4 as an evaluator to compare the two responses (Dubois et al., 2023). We report the win rate of our model compared to the baseline model (more details in Appendix D).

### 3.2 SHEARED-LLAMA OUTPERFORMS LMS OF EQUIVALENT SIZES

We demonstrate, on both standard LM benchmarks and instruction tuning, Sheared-LLaMA significantly outperforms existing LLMs of similar sizes, while using only a fraction of the compute budget to train those models from scratch.

**Downstream tasks.** In Table 2, we present the zero-shot and few-shot downstream task performance of both Sheared-LLaMA and existing pre-trained models of a similar size. Our experiments show that, even with a budget as limited as approximately 50B tokens for pruning and continued pre-training, Sheared-LLaMA models outperform existing models that have been pre-trained on significantly larger compute. To elaborate further, Sheared-LLaMA-1.3B outperforms both the OPT-1.3B and Pythia-1.4B models, which were originally pre-trained with 300B tokens. Similarly, Sheared-LLaMA-2.7B outperforms INCITE-Base-3B and OpenLLaMA-3B-v1, which were pre-trained on 800B and 1T RedPajama tokens respectively; Sheared-LLaMA-2.7B also surpasses OpenLLaMA-3B-v2, which was trained on 1T tokens from a mixture of RedPajama, RefinedWeb, and StarCoder.

**Instruction tuning.** As shown Figure 3, instruction-tuned Sheared-LLaMA achieves higher win rates compared to all the other pre-trained models at a comparable scale. This demonstrates that our 2.7B model can serve as a strong foundation for instruction tuning and has the capacity to generate long, coherent and informative responses (See examples in Appendix D).

<sup>4</sup>OPT data contains BookCorpus (Zhu et al., 2015), Stories (Trinh & Le, 2018), CCNews (Hamborg et al., 2017), the Pile (Gao et al., 2020), and PushShift.io Reddit (Baumgartner et al., 2020).

<sup>5</sup>OpenLLaMA v2 is pre-trained with a mixture of RefinedWeb (Penedo et al., 2023), StarCoder (Li et al., 2023), and part of RedPajama.

<sup>6</sup>[https://huggingface.co/spaces/HuggingFaceH4/open\\_llm\\_leaderboard](https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard)

<sup>7</sup><https://sharegpt.com>. We only use the first round in the multi-turn chat history.

Table 2: Sheared-LLaMA outperforms publicly available models of comparable size on downstream tasks. The shot number used is noted in parentheses, with 0-shot if not specified. Models with  $\dagger$  use a different training data from RedPajama. Please refer to Table 1 for details.

Model (#tokens for training)	Commonsense & Reading Comprehension					
	SciQ	PIQA	WinoGrande	ARC-E	ARC-C (25)	HellaSwag (10)
LLaMA2-7B (2T) $\dagger$	93.7	78.1	69.3	76.4	53.0	78.6
OPT-1.3B (300B) $\dagger$	84.3	71.7	<b>59.6</b>	57.0	29.7	54.5
Pythia-1.4B (300B) $\dagger$	86.4	70.9	57.4	60.7	31.2	53.0
Sheared-LLaMA-1.3B (50B)	<b>87.3</b>	<b>73.4</b>	57.9	<b>61.5</b>	<b>33.5</b>	<b>60.7</b>
OPT-2.7B (300B) $\dagger$	85.8	73.7	60.8	60.8	34.0	61.5
Pythia-2.8B (300B) $\dagger$	88.3	74.0	59.7	64.4	36.4	60.8
INCITE-Base-3B (800B)	90.7	74.6	63.5	<b>67.7</b>	40.2	64.8
Open-LLaMA-3B-v1 (1T)	91.3	73.7	61.5	67.6	39.6	62.6
Open-LLaMA-3B-v2 (1T) $\dagger$	<b>91.8</b>	<b>76.2</b>	63.5	66.5	39.0	67.6
Sheared-LLaMA-2.7B (50B)	90.8	75.8	<b>64.2</b>	67.0	<b>41.2</b>	<b>70.8</b>
Continued						
Model (#tokens for training)	LogiQA	BoolQ (32)	LAMBADA	NQ (32)	MMLU (5)	Average
	30.7	82.1	28.8	73.9	46.6	64.6
LLaMA2-7B (2T) $\dagger$	<b>26.9</b>	57.5	58.0	6.9	24.7	48.2
OPT-1.3B (300B) $\dagger$	27.3	57.4	<b>61.6</b>	6.2	<b>25.7</b>	48.9
Pythia-1.4B (300B) $\dagger$	<b>26.9</b>	<b>64.0</b>	61.0	<b>9.6</b>	<b>25.7</b>	<b>51.0</b>
OPT-2.7B (300B) $\dagger$	26.0	63.4	63.6	10.1	25.9	51.4
Pythia-2.8B (300B) $\dagger$	28.0	66.0	64.7	9.0	26.9	52.5
INCITE-Base-3B (800B)	27.7	65.9	65.3	14.9	<b>27.0</b>	54.7
Open-LLaMA-3B-v1 (1T)	28.4	70.0	65.4	<b>18.6</b>	<b>27.0</b>	55.1
Open-LLaMA-3B-v2 (1T) $\dagger$	28.1	69.6	66.5	17.1	26.9	55.7
Sheared-LLaMA-2.7B (50B)	<b>28.9</b>	<b>73.7</b>	<b>68.4</b>	16.5	26.4	<b>56.7</b>

### Comparison to further pre-training an existing LM.

We examine if pruning produces a better initialization for continued pre-training than an existing LLM of equivalent size. We continue pre-training an INCITE-Base-3B model on the original RedPajama data and compare it to Sheared-LLaMA-2.7B. Figure 4 shows that the INCITE-Base-3B model starts off with much higher accuracy, but its performance plateaus throughout continued pre-training. In contrast, Sheared-LLaMA starts at a lower accuracy but rapidly improves, eventually surpassing the INCITE-Base-3B model. This suggests that pruned models from a strong base model serve as a better initialization for continued pre-training.<sup>8</sup> Please find more training details in Appendix F.

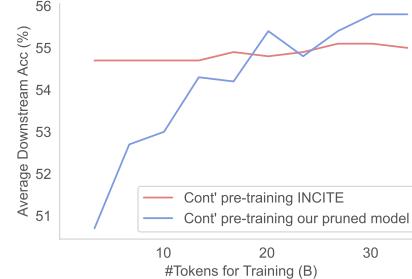


Figure 4: Average downstream performance of continuing pre-training Sheared-LLaMA vs INCITE-Base-3B.

## 4 ANALYSIS

### 4.1 EFFECTIVENESS OF DYNAMIC BATCH LOADING

We analyze the effectiveness of dynamic batch loading by examining its impact on three aspects: (1) the final LM loss across domains, (2) the data usage of each domain throughout training, (3) the downstream task performance. All results in this section are based on Sheared-LLaMA-1.3B.

<sup>8</sup>In cases where the existing small model is competitive compared to the pruning source model, the small model may offer a better starting point than a pruned model. Intuitively, the larger the discrepancy in performance between the source model and the small model, the more advantages the pruned model .

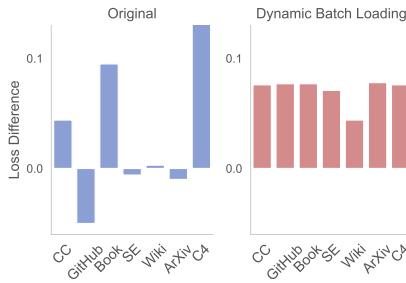


Figure 5: Loss difference between the pruned model (1.3B) and estimated reference loss, with original vs. dynamic batch loading.

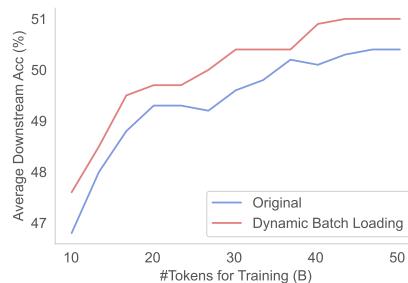


Figure 6: Downstream task performance of Sheared-LLaMA-1.3B with original data proportion and dynamic batch loading.

Table 3: Domain data usage with dynamic batch loading compared to the original proportions.

	CC	GitHub	Book	StackExchange	Wiki	ArXiv	C4
RedPajama	67.0%	4.5%	4.5%	2.0%	4.5%	2.5%	15.0%
Dynamic Batch Loading	36.1%	0.8%	9.1%	1.0%	3.1%	0.7%	49.2%

**Loss differences across domains.** Dynamic batch loading is designed to balance the rate of loss reduction across domains, so that the losses reach the reference value at approximately the same time. In Figure 5, we plot the difference between the loss of our model (with both original and dynamic batch loading) and the reference loss, estimated by fitting a scaling function to a hypothetical 2.7B parameter LLaMA2 model. With the original batch loading, the loss differences vary dramatically across domains. For instance, the GitHub loss decreases below the reference value, while the C4 loss lags behind. In contrast, dynamic batch loading reduces losses evenly and shows very similar loss differences across domains, indicating a more efficient data use.

**Data usage.** Table 3 compares the original data proportion of RedPajama and the domain data usage of our dynamic loading (Figure 7 shows the evolution of domain weights throughout the training). We see that dynamic batch loading increases the weights for the Book and C4 domains versus other domains—suggesting that they are more difficult to recover for a pruned model.

**Downstream performance.** As shown in Figure 6, pruned models trained with dynamic batch loading achieve better downstream performance than when trained on the original RedPajama distribution. This suggests that the more balanced loss reduction from dynamic batch loading transfers to improved downstream capabilities.

#### 4.2 COMPARISON TO OTHER PRUNING APPROACHES

We compare our LLM-shearing method to other pruning approaches and report validation perplexity, which serves as a strong indicator of overall model capabilities (Xia et al., 2023). Due to computational constraints, the following experiments control the total compute budget across compared methods rather than running each method to completion.

**Targeted pruned models have a higher inference throughput.** Previous works like Block Pruning (Lagunas et al., 2021) or CoFiPruning (Xia et al., 2022) are experimented on BERT-scale LMs, and the final model architectures, though structured, usually have non-uniform layer configurations, e.g., different layers have different number of heads or intermediate size. While bringing performance gains, non-uniformity also introduces training and inference overhead due to irregularities in model architectures. We experiment with both CoFiPruning and our targeted structured pruning. For a fair comparison, we use the same original data proportion for both approaches. As shown in Table 4, our targeted pruned models have a higher inference throughput compared to the non-uniformly pruned CoFiPruning model at the same sparsity, despite having slightly higher perplexity.

Table 4: Validation perplexity and inference throughout (tokens/second) of targeted structured pruning (without continued pre-training) with a uniform layer configuration, and CoFiPruning, with a non-uniform layer configuration. Inference throughput is measured on a Nvidia A100 (80G) GPU, with a batch size of 1 and a sequence length of 512.

	<b>Layer Config</b>	<b>PPL ↓</b>	<b>Throughput ↑</b>		<b>Layer Config</b>	<b>PPL ↓</b>	<b>Throughput ↑</b>
1.3B	CoFiPruning	9.1	51	3B	CoFiPruning	7.0	37
	Ours	10.3	58		Ours	7.7	43

**Comparison to LLM-Pruner (Ma et al., 2023).** We compare our pruning method to LLM-Pruner, a recent work in uniform layer configuration structured pruning, in Appendix E.2. We show that with the same budget and the compression rate, ours achieves better perplexity.

#### 4.3 ADDITIONAL ANALYSIS

**Budget allocation for pruning and continued pre-training.** Intuitively, allocating more compute to the pruning stage helps identify better subnetwork structures. We explore distributing data across pruning and continued pre-training stages differently, within a fixed budget of 5B tokens. Table 5 shows that when controlling the total amount of tokens, increasing the pruning budget consistently improves perplexity. However, since pruning is more expensive than continued pre-training, we decide to allocate 0.4B tokens to pruning. Please refer to Appendix B for details on training throughputs

Table 5: Data budget allocation to pruning and continued pre-training (CT) and corresponding perplexity.

# Tokens		<b>PPL</b>	
Pruning	CT	Pruning	CT
0.2B	4.6B	12.99	7.46
0.4B	4.4B	10.29	7.32
0.8B	4.0B	9.01	7.23
1.6B	3.2B	8.04	7.08

**Performance on math and coding tasks.** We also evaluate Sheared-LLaMA and baseline models on math and coding benchmarks in Appendix E.3. Sheared-LLaMA outperforms baselines trained on the same RedPajama data, but lags behind models trained on more ArXiv and GitHub data. This highlights a limitation of our work, where the performance is bounded by the chosen reference loss. To improve over math and coding, a better initial data proportion (e.g., more GitHub) and better reference losses are needed , and we leave it for future work.

## 5 RELATED WORK

**Pruning.** Structured pruning has been extensively studied as a model compression technique in computer vision and natural language processing, where task-specific models like classification ones are often overparameterized and can be pruned significantly with minimal impact on performance (Han et al., 2016; Wen et al., 2016; Liu et al., 2017; Luo et al., 2017; Cai et al., 2019; Deng et al., 2020; Hou et al., 2020; Wang et al., 2020; Lagunas et al., 2021; Xia et al., 2022; Kurtic et al., 2023). Unstructured pruning (Frankle & Carbin, 2018; Li et al., 2020; Chen et al., 2020; Sanh et al., 2020) prunes individual neurons instead of structured blocks. Though unstructured pruning usually achieve higher compression rates, they are not practical for model speedup.

In the era of LLMs, the prevalent NLP pipeline has shifted from task-specific models to general-purpose LMs, which leaves little room for redundancy. Both unstructured pruning, semi-structured pruning (Frantar & Alistarh, 2023; Sun et al., 2023), and structured pruning (Ma et al., 2023) lead to significant performance drops on LLM even at a modest sparsity. Noticeably, all the aforementioned works fix the original model parameters or tune them minimally. In our work, we see pruning as an initialization and consider it necessary to expend substantial compute to continually pre-training the model to recover performance.

**Efficient pre-training approaches.** As orthogonal to our pruning approach, There is an extensive body of work on improving efficiency of training LLMs. For example, quantization reduces the numeric precision of model weights and activations and speeds up training and inference (Dettmers et al., 2022; 2023; Xiao et al., 2023). Knowledge distillation (Hinton et al., 2015; Sanh et al., 2019; Jiao et al., 2020; Sun et al., 2020), which trains a smaller model on a larger model’s prediction, is

shown to be effective for task-specific models (Xia et al., 2022). For pre-training LLMs, though distilling from a teacher model is shown to improve the quality of student models given the same number of training steps (Rae et al., 2021; Blakeney et al., 2022), it is less cost-effective than pruning and continued training due to the exceeding inference cost incurred by the teacher model (Jha et al., 2023). More methods have been introduced to enhance the efficiency of training LMs, such as dynamic architectures (Gong et al., 2019; Zhang & He, 2020) and efficient optimizers (Chen et al., 2023; Liu et al., 2023). However, as indicated by (Kaddour et al., 2023), the promised gains in training efficiency may not be consistently realized.

There are also data-based approaches to enhance training efficiency. Eliminating duplicated data is found to be effective (Lee et al., 2021). Various batch selection techniques propose to prioritize data based on criteria such as higher losses (Jiang et al., 2019) or a greater reducible loss (Mindermann et al., 2022). Xie et al. (2023) propose to optimize data mixtures by training a proxy model to estimate the optimal data weight of each domain.

## 6 DISCUSSION

**Limitation and future work.** This work has two main limitations. First, the method relies heavily on the availability of open-source pre-training datasets and large language models. If the pre-training data does not cover a particular domain, the method is unlikely to recover performance well on that domain. Second, due to computational constraints, we only conducted experiments using a 7B parameter model. However, our method is highly generalizable and can be scaled up to larger models in future research.

**Conclusion.** In this work, we propose using structured pruning as an efficient approach to producing competitive LLMs. Our approach consists of two stages, *targeted structured pruning* and *continued pre-training*, and we propose *dynamic batch loading* to improve the efficiency of using pre-training data. We train a series of competitive Sheared-LLaMA models with a fraction of compute compared to standard pre-training. Our results highlight a promising avenue to produce small LLMs with a low cost when strong large-scale models are available. As more capable LLMs and larger pre-training datasets emerge, our method can easily extend to these advances to produce improved small models.

## ACKNOWLEDGEMENTS

We express our gratitude to Sadhika Malladi, Tanya Goyal, Ofir Press, Adithya Bhaskar, and the Princeton NLP group for reviewing the paper and providing helpful feedback. We also thank the engineering team at MosaicML for their invaluable assistance with implementation specifics using the Composer package. Mengzhou Xia is supported by a Bloomberg Data Science Ph.D. Fellowship, and Tianyu Gao is supported by an IBM PhD Fellowship. This research is also supported by Microsoft Azure credits through the “Accelerate Foundation Models Academic Research” Initiative.

## REFERENCES

- Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. Falcon-40B: an open large language model with state-of-the-art performance. 2023.
- Anthropic. Introducing claude, 2023.
- Jason Baumgartner, Savvas Zannettou, Brian Keegan, Megan Squire, and Jeremy Blackburn. The pushshift reddit dataset. *ArXiv*, abs/2001.08435, 2020.
- Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O’Brien, Eric Hallahan, Mohammad Aftab Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pp. 2397–2430. PMLR, 2023.

- Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Pixa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 7432–7439, 2020.
- Cody Blakeney, Jessica Zosa Forde, Jonathan Frankle, Ziliang Zong, and Matthew L Leavitt. Reduce, reuse, recycle: Improving training efficiency with distillation. *arXiv preprint arXiv:2211.00683*, 2022.
- Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2019.
- Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. The lottery ticket hypothesis for pre-trained bert networks. In *Advances in Neural Information Processing Systems*, 2020.
- Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, et al. Symbolic discovery of optimization algorithms. *arXiv preprint arXiv:2302.06675*, 2023.
- Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.
- Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.
- Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE*, 108(4): 485–532, 2020.
- Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.
- Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.
- Yann Dubois, Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. Alpacafarm: A simulation framework for methods that learn from human feedback. *arXiv preprint arXiv:2305.14387*, 2023.
- Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2018.
- Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot, 2023. *arXiv preprint arXiv:2301.00774*, 2023.
- Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.
- Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, Jason Phang, Laria Reynolds, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, September 2021.
- Xinyang Geng and Hao Liu. Openllama: An open reproduction of llama, May 2023.
- Linyuan Gong, Di He, Zhuohan Li, Tao Qin, Liwei Wang, and Tieyan Liu. Efficient training of bert by progressively stacking. In *International conference on machine learning*, pp. 2337–2346. PMLR, 2019.

- Kshitij Gupta, Benjamin Thérien, Adam Ibrahim, Mats L Richter, Quentin Anthony, Eugene Belilovsky, Irina Rish, and Timothée Lesort. Continual pre-training of large language models: How to (re) warm your model? *arXiv preprint arXiv:2308.04014*, 2023.
- Felix Hamborg, Norman Meuschke, Corinna Breitinger, and Bela Gipp. news-please: A generic news crawler and extractor. In *Proceedings of the 15th International Symposium of Information Science*, pp. 218–223, 2017.
- Song Han, Huizi Mao, Dally, and William Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations*, 2016.
- Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021.
- Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.
- Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.
- Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. Dynabert: Dynamic bert with adaptive width and depth. *Advances in Neural Information Processing Systems*, 33: 9782–9793, 2020.
- Ananya Harsh Jha, Dirk Groeneveld, Emma Strubell, and Iz Beltagy. Large language model distillation doesn't need a teacher. *arXiv preprint arXiv:2305.14864*, 2023.
- Angela H Jiang, Daniel L-K Wong, Giulio Zhou, David G Andersen, Jeffrey Dean, Gregory R Ganger, Gauri Joshi, Michael Kaminsky, Michael Kozuch, Zachary C Lipton, et al. Accelerating deep learning by focusing on the biggest losers. *arXiv preprint arXiv:1910.00762*, 2019.
- Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 4163–4174, 2020.
- Jean Kaddour, Oscar Key, Piotr Nawrot, Pasquale Minervini, and Matt J Kusner. No train no gain: Revisiting efficient training algorithms for transformer-based language models. *arXiv preprint arXiv:2307.06440*, 2023.
- Eldar Kurtic, Elias Frantar, and Dan Alistarh. Ziplm: Hardware-aware structured pruning of language models. *arXiv preprint arXiv:2302.04089*, 2023.
- Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019.
- François Lagunas, Ella Charlaix, Victor Sanh, and Alexander M Rush. Block pruning for faster transformers. *arXiv preprint arXiv:2109.04838*, 2021.
- Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. Deduplicating training data makes language models better. *arXiv preprint arXiv:2107.06499*, 2021.
- Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *International Conference on Learning Representations*, 2016.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

- Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on machine learning*, pp. 5958–5968. PMLR, 2020.
- Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training. *arXiv preprint arXiv:2305.14342*, 2023.
- Jian Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. Logiqa: A challenge dataset for machine reading comprehension with logical reasoning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pp. 3622–3628, 2020.
- Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE international conference on computer vision*, pp. 2736–2744, 2017.
- Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through 1.0 regularization. In *International Conference on Learning Representations*, 2018.
- Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pp. 5058–5066, 2017.
- Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models. *arXiv preprint arXiv:2305.11627*, 2023.
- Sören Mindermann, Jan M Brauner, Muhammed T Razzak, Mrinank Sharma, Andreas Kirsch, Winnie Xu, Benedikt Höltgen, Aidan N Gomez, Adrien Morisot, Sebastian Farquhar, et al. Prioritized training on points that are learnable, worth learning, and not yet learnt. In *International Conference on Machine Learning*, pp. 15630–15649. PMLR, 2022.
- MosaicML. composer, 2021.
- MosaicML. Introducing mpt-7b: A new standard for open-source, commercially usable llms, 2023. Accessed: 2023-05-05.
- OpenAI. Gpt-4 technical report. *ArXiv*, abs/2303.08774, 2023.
- Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35, 2022.
- Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1525–1534, 2016.
- Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon llm: outperforming curated corpora with web data, and web data only. *arXiv preprint arXiv:2306.01116*, 2023.
- Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.
- Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.
- Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.
- Victor Sanh, Thomas Wolf, and Alexander Rush. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in Neural Information Processing Systems*, 33:20378–20389, 2020.

- Noam M. Shazeer. Glu variants improve transformer. *ArXiv*, abs/2002.05202, 2020.
- Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023.
- Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 2158–2170, 2020.
- Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model, 2023.
- TogetherAI. Redpajama-incite-base-3b-v1, 2023a.
- TogetherAI. Redpajama: An open source recipe to reproduce llama training dataset, 2023b.
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.
- Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Niko lay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.
- Trieu H. Trinh and Quoc V. Le. A simple method for commonsense reasoning. *ArXiv*, abs/1806.02847, 2018.
- Peiyi Wang, Lei Li, Liang Chen, Dawei Zhu, Binghuai Lin, Yunbo Cao, Qi Liu, Tianyu Liu, and Zhipeng Sui. Large language models are not fair evaluators. *arXiv preprint arXiv:2305.17926*, 2023.
- Ziheng Wang, Jeremy Wohlwend, and Tao Lei. Structured pruning of large language models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6151–6162, 2020.
- Johannes Welbl, Nelson F. Liu, and Matt Gardner. Crowdsourcing multiple choice science questions. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*, pp. 94–106, 2017.
- Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29, 2016.
- Mengzhou Xia, Zexuan Zhong, and Danqi Chen. Structured pruning learns compact and accurate models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1513–1528, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.107.
- Mengzhou Xia, Mikel Artetxe, Chunting Zhou, Xi Victoria Lin, Ramakanth Pasunuru, Danqi Chen, Luke Zettlemoyer, and Veselin Stoyanov. Training trajectories of language models across scales. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13711–13738, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.767.
- Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pp. 38087–38099. PMLR, 2023.
- Sang Michael Xie, Hieu Pham, Xuanyi Dong, Nan Du, Hanxiao Liu, Yifeng Lu, Percy Liang, Quoc V Le, Tengyu Ma, and Adams Wei Yu. Doremi: Optimizing data mixtures speeds up language model pretraining. *arXiv preprint arXiv:2305.10429*, 2023.
- Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4791–4800, 2019.

Minjia Zhang and Yuxiong He. Accelerating training of transformer-based language models with progressive layer dropping. *Advances in Neural Information Processing Systems*, 33:14011–14023, 2020.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuhui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

Yukun Zhu, Ryan Kiros, Richard S. Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 19–27, 2015.

## A REFERENCE LOSS PREDICTED BY SCALING LAWS

The scaling law of language modeling is a function of model size  $N$  and dataset size  $D$ :

$$L(N, D) = E + \frac{A}{N^\alpha} + \frac{B}{D^\beta}$$

where  $E$  captures the loss for the true language distribution in an ideal generation process, and  $A, \alpha, B, \beta$  are scaling factors related to model scale or data size. Models in the same model family are usually trained with the same amount of tokens on the same data distribution. In this case, we need a minimum of three models to estimate the constant  $E + \frac{B}{D^\beta}$ ,  $A$  and  $\alpha$ . If the models are trained with different amount of tokens, we can estimate  $E, A, \alpha, B, \beta$  with a minimal of 5 models. Note that we will estimate the scaling factors for each domain separately.

It is known that LLAMA2 models have been trained on the same 2T tokens (Touvron et al., 2023b). Therefore, we take the LLAMA2-7B, LLAMA2-13B and LLAMA2-70B checkpoints, evaluate them on the validation set of each domain, and fit the scaling factors with the corresponding loss. Given the limited data points for estimating the scaling law constant, we recognize the projected loss of a hypothetical LLaMA-2.7B model may be biased compared to the true value. We present the predicted loss in Table 6.

Table 6: Estimated reference loss of hypothetical LLaMA2-1.3B and LLaMA2-2.7B.

	<b>CC</b>	<b>GitHub</b>	<b>Book</b>	<b>StackExchange</b>	<b>Wiki</b>	<b>ArXiv</b>	<b>C4</b>
1.3B	1.964	0.746	2.139	1.612	1.759	1.445	2.125
2.7B	1.871	0.688	2.033	1.535	1.630	1.356	2.033

## B TRAINING DETAILS

We present the hyperparameters used in our experiments in Appendix B. We use fully sharded data parallel (Zhao et al., 2023) to train our models in parallel. We use FlashAttention V1 (Dao et al., 2022) to speed up training. We use a cosine learning rate scheduler and decay the learning rate to a minimum of 10% of the peak value. We conduct some preliminary experiment to determine the peak learning rate for learning the masking variables and Lagrange multipliers, and we find that a learning rate of 1.0 works well for pruning. We do not tune any other hyper-parameters. The throughput is dependent on the implementations and we believe that our throughput can be further improved by adopting more advanced recent optimizations such as FlashAttention V2 (Dao et al., 2022) and a more recent version of Composer.

Table 7: Training hyper-parameters and throughput.

	Pruning	Continued Pre-training
Training budget	0.4B	50B
Learning rate of $z, \phi, \lambda$	1.0	-
Learning Rate of $\theta$	0.0001	0.0001
LR warmup ratio	10%	3%
Batch size (tokens)	131K	1M
Evaluation interval $m$ (steps)	50	400
Steps	3,200	51,200
# GPUs	8	16
Throughput (tokens/s)	15K	145K (1.3B) / 77K (2.7B)

## C MODEL CONFIGURATIONS

In this section, we provide the model configurations for both our Sheared-LLaMA model and the baseline models, as illustrated in Table 8. Our design closely adheres to the architecture of Pythia-1.4B and INCITE-Base-3B, albeit with some nuanced distinctions. A noteworthy difference is found

in the intermediate size of Sheared-LLaMA, which is a consequence of its lineage from LLaMA2-7B. Notably, LLaMA2-7B employs a GLU variant (Shazeer, 2020) within its feed-forward layer, comprising a gate matrix, an upward-projection matrix, and a downward-projection matrix. In contrast, other models employ the conventional double-matrix feed-forward layer structure. Furthermore, we acknowledge that the shearing algorithm will have to inherit the head dimension of the source model. Instead of explicitly specifying the number of heads based on existing language models, we set the target number of heads to be the target hidden dimension divided by the head dimension of the source model.

Table 8: Model configurations of our Sheared-LLaMA and baseline models.

Model	#Param	#Layers	Hidden	Intermediate	#Heads	Head Dim
OPT-1.3B	1.3B	24	2048	8192	32	64
Pythia-1.4B	1.4B	24	2048	8192	16	128
Sheared-LLaMA-1.3B	1.3B	24	2048	5504	16	128
OPT-2.7B	2.7B	32	2560	10240	32	80
Pythia-2.8B	2.8B	32	2560	10240	32	80
INCITE-Base-3B	2.8B	32	2560	10240	32	80
OpenLLaMA-3B	2.7B	26	3200	8640	32	100
Sheared-LLaMA-2.7B	2.7B	32	2560	6912	20	128
LLaMA2-7B	6.7B	32	4096	11008	32	128

## D INSTRUCTION TUNING

During instruction tuning training, the instruction is prepended with “You are a helpful assistant. Write a response that appropriately completes the request.”. For evaluating the instruction tuning generations, Wang et al. (2023) observes using GPT models as a judge could change its preference when swapping the presentation order of the two outputs. Therefore, we compare each output pair twice by swapping the presentation order of the two outputs and finally report the average win-rate of the two rounds to eliminate the position bias.

We randomly select an output generated by Sheared-LLaMA-1.3B and Sheared-LLaMA-2.7B in response to a given instruction, and present the generations in Table 10. Our findings demonstrate that, after instruction tuning, Sheared-LLaMA-2.7B consistently produces long, coherent, and informative outputs in response to the instruction.

Table 9: Training hyper-parameters for instruction tuning.

Instruction Tuning	
Learning Rate of $\theta$	$5e - 5$
LR warmup ratio	3%
Batch size (tokens)	128
# GPUs	8

## E ADDITIONAL RESULTS

### E.1 DATA USAGE IN CONTINUED PRE-TRAINING

Figure 7 shows how the domain weights change throughout the training process and the final cumulative data usage of each domain. The trajectory shows that the domain weights stabilize after around 30% training. Unlike other domains, Wikipedia exhibits an anomalous spike in data loading early in training. The remaining domains demonstrate a steady, monotonic change in data loading over time as expected.

Table 10: An example output from Sheared-LLaMA-1.3B and Sheared-LLaMA-2.7B. It shows that Sheared-LLaMA can generate long, coherent, and informative responses after instruction tuning.

**Prompt:** Act as semiconductor industry analyst

**Sheared-LLaMA-1.3B:** As a semiconductor industry analyst, I would be responsible for conducting market research and analysis to help companies in the industry understand and respond to market trends, customer preferences, and competitive dynamics. This would include analyzing market data such as sales, market share, and industry trends, as well as conducting interviews with industry leaders, executives, and customers to gain insights into customer needs and preferences. I would also be responsible for developing and implementing marketing strategies to promote products and services, and for identifying and pursuing new business opportunities.

**Sheared-LLaMA-2.7B:** As a semiconductor industry analyst, my primary responsibility is to provide insightful and accurate analysis on the global semiconductor market. This includes evaluating the performance of individual companies, industries, and trends within the market.

Here are some key areas where I would focus my analysis:

1. Market size and growth: I would track the size of the global semiconductor market and analyze the growth prospects for each segment, such as memory, logic, and sensors.
2. Competitive landscape: I would assess the competitive landscape, including the top players, their market share, and strategic moves.
3. Technology trends: I would monitor the development of new technologies, such as 5G, AI, and IoT, and their impact on the semiconductor industry.
4. Industry trends: I would track industry trends, such as the shift towards sustainable technologies, and evaluate their impact on the market.
5. Market forecasts: I would provide forecasts for the semiconductor market, including revenue and unit shipment growth, based on a range of scenarios and assumptions.

In addition to these areas, I would also keep track of industry news and developments, and provide insights on their impact on the market.



Figure 7: Data weight of each batch during the continued pre-training stage.

## E.2 COMPARISON TO LLM-PRUNER

Table 11 displays the model configurations for an LLM-Pruner pruned model (Ma et al., 2023) versus our pruned model. The model pruned from LLM-Pruner has an unconventional architecture where the intermediate size is smaller than hidden size, largely due to the fact that the algorithm does not support pruning the hidden dimension. When comparing performance between LLM-Pruner and ours in continued pre-training, our model achieves lower perplexity than LLM-Pruner with a similar

parameter count and the same amount of continued pre-training, demonstrating the effectiveness of the targeted structured pruning.

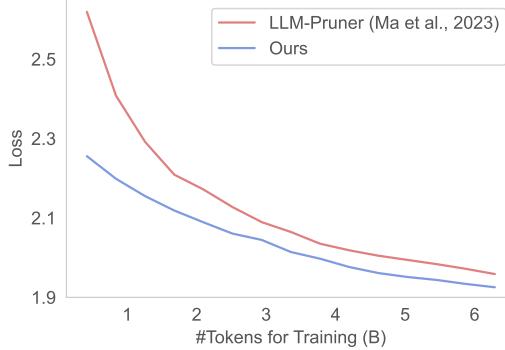


Figure 8: The loss of the 1.3B model with original loading and dynamic loading and the downstream trajectory.

Table 11: Model structure of Pythia-1.4B, LLM-pruner (1.3B), and Ours (1.3B). With a similar parameter count, our pruned model structure has a lower perplexity when fine-tuned with the same amount of tokens (around 6B tokens).

	Layers	Heads	Head size	Intermediate size	Hidden size	Params	PPL
Pythia-1.4B	24	16	128	8192	2048	1.4B	-
LLM-pruner (1.6B)	32	7	128	2201	4096	1.6B	7.09
Ours (1.3B)	24	16	128	5504	2048	1.3B	6.85

### E.3 CODING AND MATH REASONING

Table 12: Evaluation results on GSM8K and HumanEval and training percentage and tokens in ArXiv and GitHub.

Models	GSM8K (8)		HumanEval		ArXiv		GitHub		ArXiv Tokens	GitHub Tokens
	EM	Pass@1	Pass@5	Percentage	Percentage	Tokens				
LLaMA2-7B	13.7	12.8	23.8	-	-	-	-	-	-	-
OPT-2.7B	0.1	0.0	0.0	-	-	-	-	-	-	-
Pythia-2.8B	1.7	5.1	14.6	9.0%	7.6%	26.9	22.8	-	-	-
INCITE-Base-3B	1.8	4.3	4.9	2%	4.5%	16.0	36.0	-	-	-
Open-LLaMA-3B-v1	2.5	0.0	1.2	2%	4.5%	20.0	45.0	-	-	-
Open-LLaMA-3B-v2	2.7	10.4	20.1	-	-	-	-	-	-	-
Sheared-LLaMA-2.7B (Source)	2.7	3.7	5.5	0.7%	0.4%	0.3	0.2	-	-	-
Sheared-LLaMA-2.7B (Scaling)	2.4	4.9	9.2	1.0%	0.8%	0.5	0.4	-	-	-

We examine the math and coding abilities of our pruned models compared to other language models. We find that the math ability of existing 3B parameter models, including Sheared-LLaMA, is still far below that of larger models. We also find that Sheared-LLaMA’s coding ability lags behind models known to be trained on more code data, like Pythia-1.4B and Open-LLaMA-3B-v2. Sheared-LLaMA’s coding ability likely comes from the original LLaMA2 model, speculated to have used more code data, and the minimal code data used in our pruning experiments.

### E.4 SCALING REFERENCE VS. SOURCE REFERENCE

Figure 9 compares the performance of Sheared-LLaMA when trained with the scaling reference and the source reference in dynamic batch loading. While both methods are effective in efficiently training the model, the scaling reference performs consistently (slightly) better in terms of downstream performance.

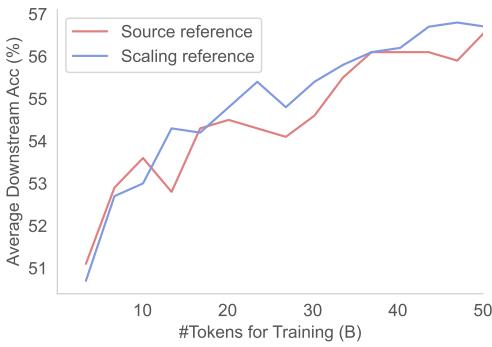


Figure 9: Average downstream performance of Sheared-LLaMA with the scaling reference and the source reference.

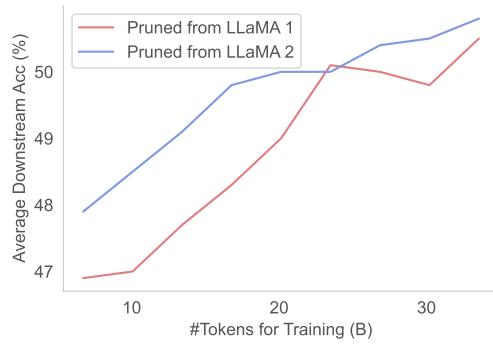


Figure 10: A comparison between pruning from LLaMA1 and LLaMA2 with dynamic loading to match the source reference loss.

## E.5 PRUNING FROM LLaMA1 VS LLaMA2

In this section, we compare the performance of pruning from LLaMA1 and LLaMA2. Both models demonstrate strong downstream task performance, though not surprisingly, pruning from LLaMA2 yields a consistent advantage.

## F TRAINING DETAILS TO CONTINUAL PRE-TRAINING INCITE-BASE-3B

Before continuing pre-training the INCITE-Base-3B model, we conduct an initial grid search to evaluate various learning rates, including values of  $1 \times 10^{-4}$ ,  $5 \times 10^{-5}$ , and  $1 \times 10^{-5}$ . Our initial results reveal that employing the first two learning rates resulted in a noticeable decline in model performance compared to the original model. Consequently, we opt to continue pre-training with a learning rate of  $1 \times 10^{-5}$ . The remaining hyperparameters remain consistent with those outlined in Appendix B. It is worth noting that our choice of continued pre-training setup may not be optimal according to recent research (Gupta et al., 2023); however, it represents the best approach within our compute constraints.

---

# LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale

---

Tim Dettmers<sup>λ\*</sup>Mike Lewis<sup>†</sup>Younes Belkada<sup>§□</sup>Luke Zettlemoyer<sup>†λ</sup>

University of Washington<sup>λ</sup>  
 Facebook AI Research<sup>†</sup>  
 Hugging Face<sup>§</sup>  
 ENS Paris-Saclay<sup>□</sup>

## Abstract

Large language models have been widely adopted but require significant GPU memory for inference. We develop a procedure for Int8 matrix multiplication for feed-forward and attention projection layers in transformers, which cut the memory needed for inference by half while retaining full precision performance. With our method, a 175B parameter 16/32-bit checkpoint can be loaded, converted to Int8, and used immediately without performance degradation. This is made possible by understanding and working around properties of highly systematic emergent features in transformer language models that dominate attention and transformer predictive performance. To cope with these features, we develop a two-part quantization procedure, **LLM.int8()**. We first use vector-wise quantization with separate normalization constants for each inner product in the matrix multiplication, to quantize most of the features. However, for the emergent outliers, we also include a new mixed-precision decomposition scheme, which isolates the outlier feature dimensions into a 16-bit matrix multiplication while still more than 99.9% of values are multiplied in 8-bit. Using **LLM.int8()**, we show empirically it is possible to perform inference in LLMs with up to 175B parameters without any performance degradation. This result makes such models much more accessible, for example making it possible to use OPT-175B/BLOOM on a single server with consumer GPUs. We open source our software.

## 1 Introduction

Large pretrained language models are widely adopted in NLP (Vaswani et al., 2017; Radford et al., 2019; Brown et al., 2020; Zhang et al., 2022) but require significant memory for inference. For large transformer language models at and beyond 6.7B parameters, the feed-forward and attention projection layers and their matrix multiplication operations are responsible for 95%<sup>2</sup> of consumed parameters and 65-85% of all computation (Ilharco et al., 2020). One way to reduce the size of the parameters is to quantize them to less bits and use low-bit-precision matrix multiplication. With this goal in mind, 8-bit quantization methods for transformers have been developed (Chen et al., 2020; Lin et al., 2020; Zafrir et al., 2019; Shen et al., 2020). While these methods reduce memory use, they degrade performance, usually require tuning quantization further after training, and have only been studied for models with less than 350M parameters. Degradation-free quantization up to 350M parameters is poorly understood, and multi-billion parameter quantization remains an open challenge.

---

\*Majority of research done as a visiting researcher at Facebook AI Research.

<sup>2</sup>Other parameters come mostly from the embedding layer. A tiny amount comes from norms and biases.

In this paper, we present the first multi-billion-scale Int8 quantization procedure for transformers that does not incur any performance degradation. Our procedure makes it possible to load a 175B parameter transformer with 16 or 32-bit weights, convert the feed-forward and attention projection layers to 8-bit, and use the resulting model immediately for inference without any performance degradation. We achieve this result by solving two key challenges: the need for higher quantization precision at scales beyond 1B parameters and the need to explicitly represent the sparse but systematic large magnitude outlier features that ruin quantization precision once they emerge in *all* transformer layers starting at scales of 6.7B parameters. This loss of precision is reflected in C4 evaluation perplexity (Section 3) as well as zeroshot accuracy as soon as these outlier features emerge, as shown in Figure 1.

We show that with the first part of our method, vector-wise quantization, it is possible to retain performance at scales up to 2.7B parameters. For vector-wise quantization, matrix multiplication can be seen as a sequence of independent inner products of row and column vectors. As such, we can use a separate quantization normalization constant for each inner product to improve quantization precision. We can recover the output of the matrix multiplication by de-normalizing by the outer product of column and row normalization constants before we perform the next operation.

To scale beyond 6.7B parameters without performance degradation, it is critical to understand the emergence of extreme outliers in the feature dimensions of the hidden states during inference. To this end, we provide a new descriptive analysis which shows that large features with magnitudes up to 20x larger than in other dimensions first appear in about 25% of all transformer layers and then gradually spread to other layers as we scale transformers to 6B parameters. At around 6.7B parameters, a phase shift occurs, and *all* transformer layers and 75% of all sequence dimensions are affected by extreme magnitude features. These outliers are highly systematic: at the 6.7B scale, 150,000 outliers occur per sequence, but they are concentrated in only 6 feature dimensions across the entire transformer. Setting these outlier feature dimensions to zero decreases top-1 attention softmax probability mass by more than 20% and degrades validation perplexity by 600-1000% despite them only making up about 0.1% of all input features. In contrast, removing the same amount of random features decreases the probability by a maximum of 0.3% and degrades perplexity by about 0.1%.

To support effective quantization with such extreme outliers, we develop mixed-precision decomposition, the second part of our method. We perform 16-bit matrix multiplication for the outlier feature dimensions and 8-bit matrix multiplication for the other 99.9% of the dimensions. We name the combination of vector-wise quantization and mixed precision decomposition, **LLM.int8()**. We show that by using **LLM.int8()**, we can perform inference in LLMs with up to 175B parameters without any performance degradation. Our method not only provides new insights into the effects of these outliers on model performance but also makes it possible for the first time to use very large models, for example, OPT-175B/BLOOM, on a single server with consumer GPUs. While our work focuses on making large language models accessible without degradation, we also show in Appendix D that we maintain end-to-end inference runtime performance for large models, such as BLOOM-176B and provide modest matrix multiplication speedups for GPT-3 models of size 6.7B parameters or larger. We open-source our software<sup>3</sup> and release a Hugging Face Transformers (Wolf et al., 2019) integration making our method available to all hosted Hugging Face Models that have linear layers.

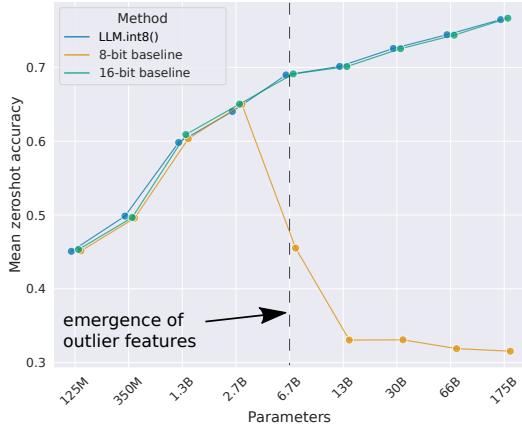


Figure 1: OPT model mean zeroshot accuracy for WinoGrande, HellaSwag, PIQA, and LAMBADA datasets. Shown is the 16-bit baseline, the most precise previous 8-bit quantization method as a baseline, and our new 8-bit quantization method, **LLM.int8()**. We can see once systematic outliers occur at a scale of 6.7B parameters, regular quantization methods fail, while **LLM.int8()** maintains 16-bit accuracy.

<sup>3</sup><https://github.com/TimDettmers/bitsandbytes>

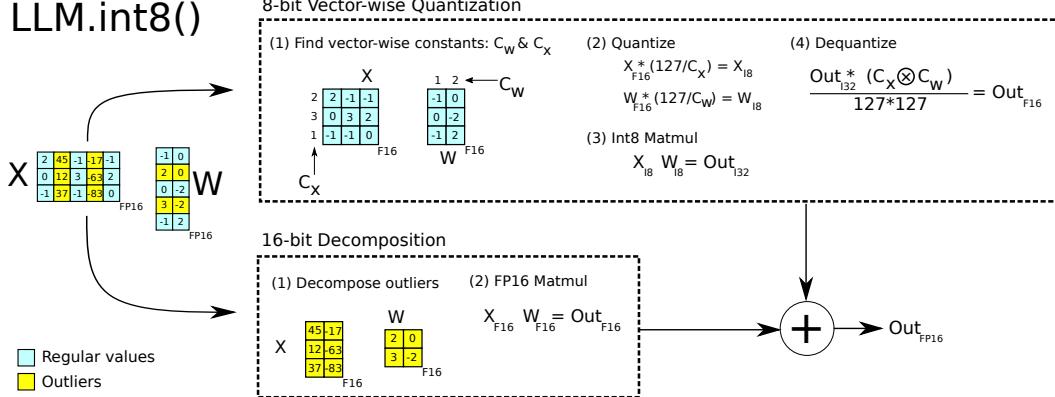


Figure 2: Schematic of LLM.int8(). Given 16-bit floating-point inputs  $\mathbf{X}_{f16}$  and weights  $\mathbf{W}_{f16}$ , the features and weights are decomposed into sub-matrices of large magnitude features and other values. The outlier feature matrices are multiplied in 16-bit. All other values are multiplied in 8-bit. We perform 8-bit vector-wise multiplication by scaling by row and column-wise absolute maximum of  $\mathbf{C}_x$  and  $\mathbf{C}_w$  and then quantizing the outputs to Int8. The Int32 matrix multiplication outputs  $\mathbf{Out}_{i32}$  are dequantized by the outer product of the normalization constants  $\mathbf{C}_x \otimes \mathbf{C}_w$ . Finally, both outlier and regular outputs are accumulated in 16-bit floating point outputs.

## 2 Background

In this work, push quantization techniques to their breaking point by scaling transformer models. We are interested in two questions: at which scale and why do quantization techniques fail and how does this relate to quantization precision? To answer these questions we study high-precision asymmetric quantization (zeropoint quantization) and symmetric quantization (absolute maximum quantization). While zeropoint quantization offers high precision by using the full bit-range of the datatype, it is rarely used due to practical constraints. Absolute maximum quantization is the most commonly used technique.

### 2.1 8-bit Data Types and Quantization

**Absmax quantization** scales inputs into the 8-bit range  $[-127, 127]$  by multiplying with  $s_{x_{f16}}$  which is 127 divided by the absolute maximum of the entire tensor. This is equivalent to dividing by the infinity norm and multiplying by 127. As such, for an FP16 input matrix  $\mathbf{X}_{f16} \in \mathbb{R}^{s \times h}$  Int8 absmax quantization is given by:

$$\mathbf{X}_{i8} = \left\lfloor \frac{127 \cdot \mathbf{X}_{f16}}{\max_{ij}(|\mathbf{X}_{f16}{}_{ij}|)} \right\rfloor = \left\lfloor \frac{127}{\|\mathbf{X}_{f16}\|_\infty} \mathbf{X}_{f16} \right\rfloor = \left\lfloor s_{x_{f16}} \mathbf{X}_{f16} \right\rfloor,$$

where  $\lfloor \cdot \rfloor$  indicates rounding to the nearest integer.

**Zeropoint quantization** shifts the input distribution into the full range  $[-127, 127]$  by scaling with the normalized dynamic range  $nd_x$  and then shifting by the zeropoint  $zp_x$ . With this affine transformation, any input tensors will use all bits of the data type, thus *reducing the quantization error for asymmetric distributions*. For example, for ReLU outputs, in absmax quantization all values in  $[-127, 0)$  go unused, whereas in zeropoint quantization the full  $[-127, 127]$  range is used. Zeropoint quantization is given by the following equations:

$$nd_{x_{f16}} = \frac{2 \cdot 127}{\max_{ij}(\mathbf{X}_{f16}{}_{ij}) - \min_{ij}(\mathbf{X}_{f16}{}_{ij})} \quad (1)$$

$$zp_{x_{i16}} = \lfloor \mathbf{X}_{f16} \cdot \min_{ij}(\mathbf{X}_{f16}{}_{ij}) \rfloor \quad (2)$$

$$\mathbf{X}_{i8} = \lfloor nd_{x_{f16}} \mathbf{X}_{f16} \rfloor \quad (3)$$

To use zeropoint quantization in an operation we feed both the tensor  $\mathbf{X}_{i8}$  and the zeropoint  $zp_{x_{i16}}$  into a special instruction<sup>4</sup> which adds  $zp_{x_{i16}}$  to each element of  $\mathbf{X}_{i8}$  before performing a 16-bit integer operation. For example, to multiply two zeropoint quantized numbers  $A_{i8}$  and  $B_{i8}$  along with their zeropoints  $zp_{a_{i16}}$  and  $zp_{b_{i16}}$  we calculate:

$$C_{i32} = \text{multiply}_{i16}(A_{zp_{a_{i16}}}, B_{zp_{b_{i16}}}) = (A_{i8} + zp_{a_{i16}})(B_{i8} + zp_{b_{i16}}) \quad (4)$$

where unrolling is required if the instruction  $\text{multiply}_{i16}$  is not available such as on GPUs or TPUs:

$$C_{i32} = A_{i8}B_{i8} + A_{i8}zp_{b_{i16}} + B_{i8}zp_{a_{i16}} + zp_{a_{i16}}zp_{b_{i16}}, \quad (5)$$

where  $A_{i8}B_{i8}$  is computed with Int8 precision while the rest is computed in Int16/32 precision. As such, zeropoint quantization can be slow if the  $\text{multiply}_{i16}$  instruction is not available. In both cases, the outputs are accumulated as a 32-bit integer  $C_{i32}$ . To dequantize  $C_{i32}$ , we divide by the scaling constants  $nd_{a_{f16}}$  and  $nd_{b_{f16}}$ .

**Int8 Matrix Multiplication with 16-bit Float Inputs and Outputs.** Given hidden states  $\mathbf{X}_{f16} \in \mathbb{R}^{s \times h}$  and weights  $\mathbf{W}_{f16} \in \mathbb{R}^{h \times o}$  with sequence dimension  $s$ , feature dimension  $h$ , and output dimension  $o$  we perform 8-bit matrix multiplication with 16-bit inputs and outputs as follows:

$$\begin{aligned} \mathbf{X}_{f16}\mathbf{W}_{f16} = \mathbf{C}_{f16} &\approx \frac{1}{c_{x_{f16}}c_{w_{f16}}} \mathbf{C}_{i32} = S_{f16} \cdot \mathbf{C}_{i32} \\ &\approx S_{f16} \cdot \mathbf{A}_{i8}\mathbf{B}_{i8} = S_{f16} \cdot Q(\mathbf{A}_{f16}) Q(\mathbf{B}_{f16}), \end{aligned} \quad (6)$$

Where  $Q(\cdot)$  is either absmax or zeropoint quantization and  $c_{x_{f16}}$  and  $c_{w_{f16}}$  are the respective tensor-wise scaling constants  $s_x$  and  $s_w$  for absmax or  $nd_x$  and  $nd_w$  for zeropoint quantization.

### 3 Int8 Matrix Multiplication at Scale

The main challenge with quantization methods that use a single scaling constant per tensor is that a single outlier can reduce the quantization precision of all other values. As such, it is desirable to have multiple scaling constants per tensor, such as block-wise constants (Dettmers et al., 2022), so that the effect of that outliers is confined to each block. We improve upon one of the most common ways of blocking quantization, row-wise quantization (Khudia et al., 2021), by using vector-wise quantization, as described in more detail below.

To handle the large magnitude outlier features that occur in all transformer layers beyond the 6.7B scale, vector-wise quantization is no longer sufficient. For this purpose, we develop mixed-precision decomposition, where the small number of large magnitude feature dimensions ( $\approx 0.1\%$ ) are represented in 16-bit precision while the other 99.9% of values are multiplied in 8-bit. Since most entries are still represented in low-precision, we retain about 50% memory reduction compared to 16-bit. For example, for BLOOM-176B, we reduce the memory footprint of the model by 1.96x.

Vector-wise quantization and mixed-precision decomposition are shown in Figure 2. The **LLM.int8()** method is the combination of absmax vector-wise quantization and mixed precision decomposition.

#### 3.1 Vector-wise Quantization

One way to increase the number of scaling constants for matrix multiplication is to view matrix multiplication as a sequence of independent inner products. Given the hidden states  $\mathbf{X}_{f16} \in \mathbb{R}^{b \times h}$  and weight matrix  $\mathbf{W}_{f16} \in \mathbb{R}^{h \times o}$ , we can assign a different scaling constant  $c_{x_{f16}}$  to each row of  $\mathbf{X}_{f16}$  and  $c_w$  to each column of  $\mathbf{W}_{f16}$ . To dequantize, we denormalize each inner product result by  $1/(c_{x_{f16}}c_{w_{f16}})$ . For the whole matrix multiplication this is equivalent to denormalization by the outer product  $\mathbf{c}_{x_{f16}} \otimes \mathbf{c}_{w_{f16}}$ , where  $\mathbf{c}_x \in \mathbb{R}^s$  and  $\mathbf{c}_w \in \mathbb{R}^o$ . As such the full equation for matrix multiplication with row and column constants is given by:

$$\mathbf{C}_{f16} \approx \frac{1}{\mathbf{c}_{x_{f16}} \otimes \mathbf{c}_{w_{f16}}} \mathbf{C}_{i32} = S \cdot \mathbf{C}_{i32} = S \cdot \mathbf{A}_{i8}\mathbf{B}_{i8} = S \cdot Q(\mathbf{A}_{f16}) Q(\mathbf{B}_{f16}), \quad (7)$$

which we term *vector-wise quantization* for matrix multiplication.

---

<sup>4</sup><https://www.felixcloutier.com/x86/pmaddubswo>

### 3.2 The Core of LLM.int8(): Mixed-precision Decomposition

In our analysis, we demonstrate that a significant problem for billion-scale 8-bit transformers is that they have large magnitude features (*columns*), which are important for transformer performance and require high precision quantization. However, vector-wise quantization, our best quantization technique, quantizes each *row* for the hidden state, which is ineffective for outlier features. Luckily, we see that these outlier features are both incredibly sparse and systematic in practice, making up only about 0.1% of all feature dimensions, thus allowing us to develop a new decomposition technique that focuses on high precision multiplication for these particular dimensions.

We find that given input matrix  $\mathbf{X}_{f16} \in \mathbb{R}^{s \times h}$ , these outliers occur systematically for almost all sequence dimensions  $s$  but are limited to specific feature/hidden dimensions  $h$ . As such, we propose *mixed-precision decomposition* for matrix multiplication where we separate outlier feature dimensions into the set  $O = \{i | i \in \mathbb{Z}, 0 \leq i \leq h\}$ , which contains all dimensions of  $h$  which have at least one outlier with a magnitude larger than the threshold  $\alpha$ . In our work, we find that  $\alpha = 6.0$  is sufficient to reduce transformer performance degradation close to zero. Using Einstein notation where all indices are superscripts, given the weight matrix  $\mathbf{W}_{f16} \in \mathbb{R}^{h \times o}$ , mixed-precision decomposition for matrix multiplication is defined as follows:

$$\mathbf{C}_{f16} \approx \sum_{h \in O} \mathbf{X}_{f16}^h \mathbf{W}_{f16}^h + \mathbf{S}_{f16} \cdot \sum_{h \notin O} \mathbf{X}_{i8}^h \mathbf{W}_{i8}^h \quad (8)$$

where  $\mathbf{S}_{f16}$  is the denormalization term for the Int8 inputs and weight matrices  $\mathbf{X}_{i8}$  and  $\mathbf{W}_{i8}$ .

This separation into 8-bit and 16-bit allows for high-precision multiplication of outliers while using memory-efficient matrix multiplication with 8-bit weights of more than 99.9% of values. Since the number of outlier feature dimensions is not larger than 7 ( $|O| \leq 7$ ) for transformers up to 13B parameters, this decomposition operation only consumes about 0.1% additional memory.

### 3.3 Experimental Setup

We measure the robustness of quantization methods as we scale the size of several publicly available pretrained language models up to 175B parameters. The key question is not how well a quantization method performs for a particular model but the trend of how such a method performs as we scale.

We use two setups for our experiments. One is based on language modeling perplexity, which we find to be a highly robust measure that is very sensitive to quantization degradation. We use this setup to compare different quantization baselines. Additionally, we evaluate zeroshot accuracy degradation on OPT models for a range of different end tasks, where we compare our methods with a 16-bit baseline.

For the language modeling setup, we use dense autoregressive transformers pretrained in fairseq (Ott et al., 2019) ranging between 125M and 13B parameters. These transformers have been pretrained on Books (Zhu et al., 2015), English Wikipedia, CC-News (Nagel, 2016), OpenWebText (Gokaslan and Cohen, 2019), CC-Stories (Trinh and Le, 2018), and English CC100 (Wenzek et al., 2020). For more information on how these pretrained models are trained, see Artetxe et al. (2021).

To evaluate the language modeling degradation after Int8 quantization, we evaluate the perplexity of the 8-bit transformer on validation data of the C4 corpus (Raffel et al., 2019) which is a subset of the Common Crawl corpus.<sup>5</sup> We use NVIDIA A40 GPUs for this evaluation.

To measure degradation in zeroshot performance, we use OPT models (Zhang et al., 2022), and we evaluate these models on the EleutherAI language model evaluation harness (Gao et al., 2021).

### 3.4 Main Results

The main language modeling perplexity results on the 125M to 13B Int8 models evaluated on the C4 corpus can be seen in Table 1. We see that absmax, row-wise, and zeropoint quantization fail as we scale, where models after 2.7B parameters perform worse than smaller models. Zeropoint quantization fails instead beyond 6.7B parameters. Our method, LLM.int8(), is the only method that preserves perplexity. As such, LLM.int8() is the only method with a favorable scaling trend.

<sup>5</sup><https://commoncrawl.org/>

Table 1: C4 validation perplexities of quantization methods for different transformer sizes ranging from 125M to 13B parameters. We see that absmax, row-wise, zeropoint, and vector-wise quantization leads to significant performance degradation as we scale, particularly at the 13B mark where 8-bit 13B perplexity is worse than 8-bit 6.7B perplexity. If we use LLM.int8(), we recover full perplexity as we scale. Zeropoint quantization shows an advantage due to asymmetric quantization but is no longer advantageous when used with mixed-precision decomposition.

Parameters	125M	1.3B	2.7B	6.7B	13B
32-bit Float	25.65	15.91	14.43	13.30	12.45
Int8 absmax	87.76	16.55	15.11	14.59	19.08
Int8 zeropoint	56.66	16.24	14.76	13.49	13.94
Int8 absmax row-wise	30.93	17.08	15.24	14.13	16.49
Int8 absmax vector-wise	35.84	16.82	14.98	14.13	16.48
Int8 zeropoint vector-wise	25.72	15.94	14.36	13.38	13.47
Int8 absmax row-wise + decomposition	30.76	16.19	14.65	13.25	12.46
Absmax LLM.int8() (vector-wise + decomp)	25.83	15.93	14.44	<b>13.24</b>	<b>12.45</b>
Zeropoint LLM.int8() (vector-wise + decomp)	<b>25.69</b>	<b>15.92</b>	<b>14.43</b>	<b>13.24</b>	<b>12.45</b>

When we look at the scaling trends of zero-shot performance of OPT models on the EleutherAI language model evaluation harness in Figure 1, we see that LLM.int8() maintains full 16-bit performance as we scale from 125M to 175B parameters. On the other hand, the baseline, 8-bit absmax vector-wise quantization, scales poorly and degenerates into random performance.

Although our primary focus is on saving memory, we also measured the run time of LLM.int8(). The quantization overhead can slow inference for models with less than 6.7B parameters, as compared to a FP16 baseline. However, models of 6.7B parameters or less fit on most GPUs and quantization is less needed in practice. LLM.int8() run times is about two times faster for large matrix multiplications equivalent to those in 175B models. Appendix D provides more details on these experiments.

## 4 Emergent Large Magnitude Features in Transformers at Scale

As we scale transformers, outlier features with large magnitudes emerge and strongly affect *all* layers and their quantization. Given a hidden state  $\mathbf{X} \in \mathbb{R}^{s \times h}$  where  $s$  is the sequence/token dimension and  $h$  the hidden/feature dimension, we define a feature to be a particular dimension  $h_i$ . Our analysis looks at a particular feature dimension  $h_i$  across all layers of a given transformer.

We find that outlier features strongly affect attention and the overall predictive performance of transformers. While up to 150k outliers exist per 2048 token sequence for a 13B model, these outlier features are highly systematic and only representing at most 7 unique feature dimensions  $h_i$ . Insights from this analysis were critical to developing mixed-precision decomposition. Our analysis explains the advantages of zeropoint quantization and why they disappear with the use of mixed-precision decomposition and the quantization performance of small vs. large models.

### 4.1 Finding Outlier Features

The difficulty with the quantitative analysis of emergent phenomena is two-fold. We aim to select a small subset of features for analysis such that the results are intelligible and not too complex while also capturing important probabilistic and structured patterns. We use an empirical approach to find these constraints. We define outliers according to the following criteria: the magnitude of the feature is at least 6.0, affects at least 25% of layers, and affects at least 6% of the sequence dimensions.

More formally, given a transformer with  $L$  layers and hidden state  $\mathbf{X}_l \in \mathbb{R}^{s \times h}$ ,  $l = 0 \dots L$  where  $s$  is the sequence dimension and  $h$  the feature dimension, we define a feature to be a particular dimension  $h_i$  in any of the hidden states  $\mathbf{X}_{l,i}$ . We track dimensions  $h_i$ ,  $0 \leq i \leq h$ , which have at least one value with a magnitude of  $\alpha \geq 6$  and we only collect statistics if these outliers occur in the *same* feature dimension  $h_i$  in at least 25% of transformer layers  $0 \dots L$  and appear in at least 6% of all sequence dimensions  $s$  across all hidden states  $\mathbf{X}_l$ . Since feature outliers only occur in attention projection

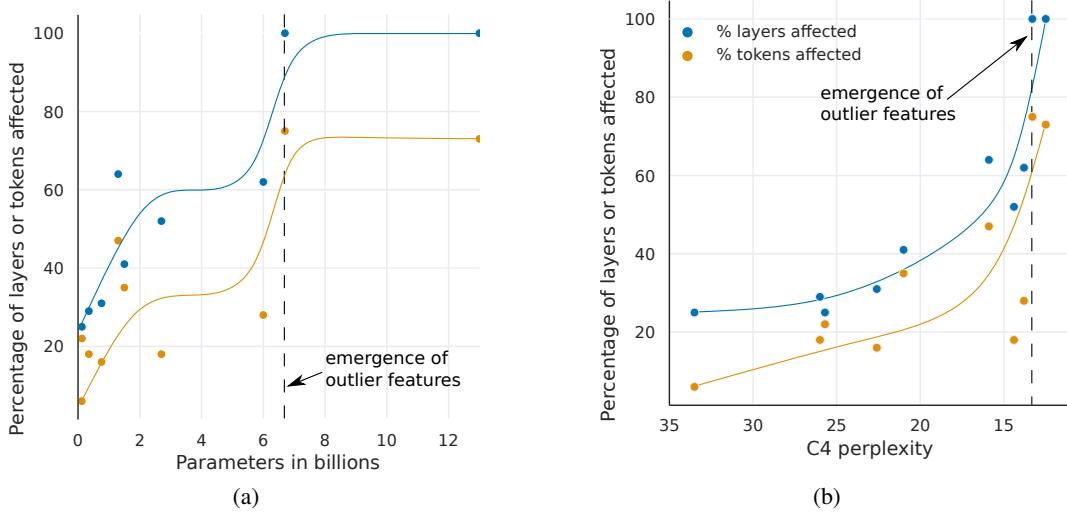


Figure 3: Percentage of layers and all sequence dimensions affected by large magnitude outlier features across the transformer by (a) model size or (b) C4 perplexity. Lines are B-spline interpolations of 4 and 9 linear segments for (a) and (b). Once the phase shift occurs, outliers are present in all layers and in about 75% of all sequence dimensions. While (a) suggest a sudden phase shift in parameter size, (b) suggests a gradual exponential phase shift as perplexity decreases. The stark shift in (a) co-occurs with the sudden degradation of performance in quantization methods.

(key/query/value/output) and the feedforward network expansion layer (first sub-layer), we ignore the attention function and the FFN contraction layer (second sub-layer) for this analysis.

Our reasoning for these thresholds is as follows. We find that using mixed-precision decomposition, perplexity degradation stops if we treat any feature with a magnitude 6 or larger as an outlier feature. For the number of layers affected by outliers, we find that outlier features are *systematic* in large models: they either occur in most layers or not at all. On the other hand, they are *probabilistic* in small models: they occur *sometimes* in *some* layers for each sequence. As such, we set our threshold for how many layers need to be affected to detect an outlier feature in such a way as to limit detection to a *single* outlier in our smallest model with 125M parameters. This threshold corresponds to that at least 25% of transformer layers are affected by an outlier in the same feature dimension. The second most common outlier occurs in only a single layer ( 2% of layers), indicating that this is a reasonable threshold. We use the same procedure to find the threshold for how many sequence dimensions are affected by outlier features in our 125M model: outliers occur in at least 6% of sequence dimensions.

We test models up to a scale of 13B parameters. To make sure that the observed phenomena are not due to bugs in software, we evaluate transformers that were trained in three different software frameworks. We evaluate four GPT-2 models which use OpenAI software, five Meta AI models that use Fairseq (Ott et al., 2019), and one EleutherAI model GPT-J that uses Tensorflow-Mesh (Shazeer et al., 2018). More details can be found in Appendix C. We also perform our analysis in two different inference software frameworks: Fairseq and Hugging Face Transformers (Wolf et al., 2019).

## 4.2 Measuring the Effect of Outlier Features

To demonstrate that the outlier features are essential for attention and predictive performance, we set the outlier features to zero before feeding the hidden states  $\mathbf{X}_l$  into the attention projection layers and then compare the top-1 softmax probability with the regular softmax probability with outliers. We do this for all layers independently, meaning we forward the regular softmax probabilities values to avoid cascading errors and isolate the effects due to the outlier features. We also report the perplexity degradation if we remove the outlier feature dimension (setting them to zero) and propagate these altered, hidden states through the transformer. As a control, we apply the same procedure for random non-outlier feature dimensions and note attention and perplexity degradation.

Our main quantitative results can be summarized as four main points.

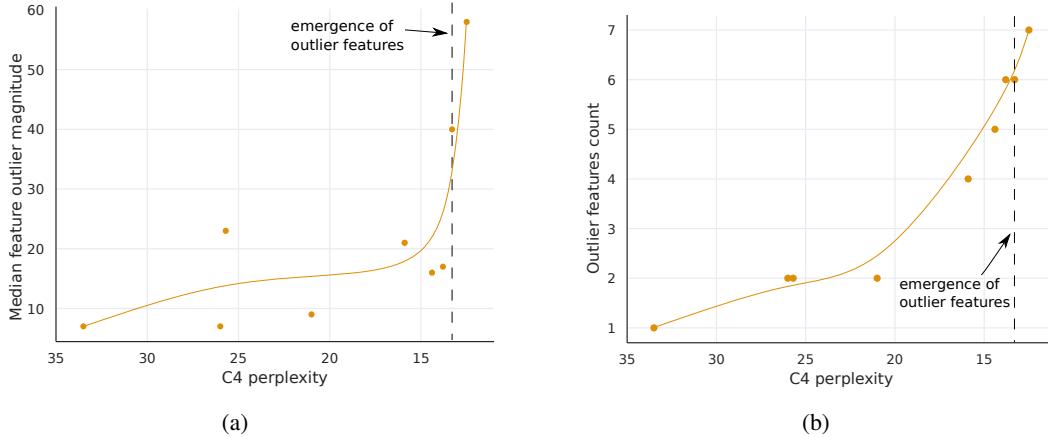


Figure 4: The median magnitude of the largest outlier feature in (a) indicates a sudden shift in outlier size. This appears to be the prime reason why quantization methods fail after emergence. While the number of outlier feature dimensions is only roughly proportional to model size, (b) shows that the number of outliers is *strictly monotonic* with respect to perplexity across all models analyzed. Lines are B-spline interpolations of 9 linear segments.

(1) When measured by the number of parameters, the emergence of large magnitude features across *all* layers of a transformer occurs suddenly between 6B and 6.7B parameters as shown in Figure 3a as the percentage of layers affected increases from 65% to 100%. The number of sequence dimensions affected increases rapidly from 35% to 75%. This sudden shift co-occurs with the point where quantization begins to fail.

(2) Alternatively, when measured by perplexity, the emergence of large magnitude features across all layers of the transformer can be seen as emerging smoothly according to an exponential function of decreasing perplexity, as seen in Figure 3b. This indicates that there is nothing sudden about emergence and that we might be able to detect emergent features before a phase shift occurs by studying exponential trends in smaller models. This also suggests that emergence is not only about model size but about perplexity, which is related to multiple additional factors such as the amount of training data used, and data quality (Hoffmann et al., 2022; Henighan et al., 2020).

(3) Median outlier feature magnitude rapidly increases once outlier features occur in all layers of the transformer, as shown in Figure 4a. The large magnitude of outliers features and their asymmetric distribution disrupts Int8 quantization precision. This is the core reason why quantization methods fail starting at the 6.7B scale – the range of the quantization distribution is too large so that most quantization bins are empty and small quantization values are quantized to zero, essentially extinguishing information. We hypothesize that besides Int8 inference, regular 16-bit floating point training becomes unstable due to outliers beyond the 6.7B scale – it is easy to exceed the maximum 16-bit value 65535 by chance if you multiply by vectors filled with values of magnitude 60.

(4) The number of outliers features increases strictly monotonically with respect to decreasing C4 perplexity as shown in Figure 4b, while a relationship with model size is non-monotonic. This indicates that model perplexity rather than mere model size determines the phase shift. We hypothesize that model size is only one important covariate among many that are required to reach emergence.

These outliers features are highly systematic after the phase shift occurred. For example, for a 6.7B transformer with a sequence length of 2048, we find about 150k outlier features per sequence for the entire transformer, but these features are concentrated in only 6 different hidden dimensions.

These outliers are critical for transformer performance. If the outliers are removed, the mean top-1 softmax probability is reduced from about 40% to about 20%, and validation perplexity increases by 600-1000% even though there are at most 7 outlier feature dimensions. When we remove 7 random feature dimensions instead, the top-1 probability decreases only between 0.02-0.3%, and perplexity increases by 0.1%. This highlights the critical nature of these feature dimensions. Quantization precision for these outlier features is paramount as even tiny errors greatly impact model performance.

### 4.3 Interpretation of Quantization Performance

Our analysis shows that outliers in particular feature dimensions are ubiquitous in large transformers, and these feature dimensions are critical for transformer performance. Since row-wise and vector-wise quantization scale each hidden state sequence dimension  $s$  (rows) and because outliers occur in the feature dimension  $h$  (columns), both methods cannot deal with these outliers effectively. This is why absmax quantization methods fail quickly after emergence.

However, almost all outliers have a strict asymmetric distribution: they are either solely positive or negative (see Appendix C). This makes zeropoint quantization particularly effective for these outliers, as zeropoint quantization is an asymmetric quantization method that scales these outliers into the full  $[-127, 127]$  range. This explains the strong performance in our quantization scaling benchmark in Table 1. However, at the 13B scale, even zeropoint quantization fails due to accumulated quantization errors and the quick growth of outlier magnitudes, as seen in Figure 4a.

If we use our full LLM.int8() method with mixed-precision decomposition, the advantage of zeropoint quantization disappears indicating that the remaining decomposed features are symmetric. However, vector-wise still has an advantage over row-wise quantization, indicating that the enhanced quantization precision of the model weights is needed to retain full precision predictive performance.

## 5 Related work

There is closely related work on quantization data types and quantization of transformers, as described below. Appendix B provides further related work on quantization of convolutional networks.

**8-bit Data Types.** Our work studies quantization techniques surrounding the Int8 data type, since it is currently the only 8-bit data type supported by GPUs. Other common data types are fixed point or floating point 8-bit data types (FP8). These data types usually have a sign bit and different exponent and fraction bit combinations. For example, a common variant of this data type has 5 bits for the exponent and 2 bits for the fraction (Wang et al., 2018; Sun et al., 2019; Cambier et al., 2020; Mellemundi et al., 2019) and uses either no scaling constants or zeropoint scaling. These data types have large errors for large magnitude values since they have only 2 bits for the fraction but provide high accuracy for small magnitude values. Jin et al. (2022) provide an excellent analysis of when certain fixed point exponent/fraction bit widths are optimal for inputs with a particular standard deviation. We believe FP8 data types offer superior performance compared to the Int8 data type, but currently, neither GPUs nor TPUs support this data type.

**Outlier Features in Language Models.** Large magnitude outlier features in language models have been studied before (Timkey and van Schijndel, 2021; Bondarenko et al., 2021; Wei et al., 2022; Luo et al., 2021). Previous work proved the theoretical relationship between outlier appearance in transformers and how it relates to layer normalization and the token frequency distribution (Gao et al., 2019). Similarly, Kovaleva et al. (2021) attribute the appearance of outliers in BERT model family to LayerNorm, and Puccetti et al. (2022) show empirically that outlier emergence is related to the frequency of tokens in the training distribution. We extend this work further by showing how the scale of autoregressive models relates to the emergent properties of these outlier features, and showing how appropriately modeling outliers is critical to effective quantization.

**Multi-billion Scale Transformer Quantization.** There are two methods that were developed in parallel to ours: nuQmm (Park et al., 2022) and ZeroQuant (Yao et al., 2022). Both use the same quantization scheme: group-w2ise quantization, which has even finer quantization normalization constant granularity than vector-wise quantization. This scheme offers higher quantization precision but also requires custom CUDA kernels. Both nuQmm and ZeroQuant aim to accelerate inference and reduce the memory footprint while we focus on preserving predictive performance under an 8-bit memory footprint. The largest models that nuQmm and ZeroQuant evaluate are 2.7B and 20B parameter transformers, respectively. ZeroQuant achieves zero-degradation performance for 8-bit quantization of a 20B model. We show that our method allows for zero-degradation quantization of models up to 176B parameters. Both nuQmm and ZeroQuant suggest that finer quantization granularity can be an effective means to quantize large models. These methods are complementary with LLM.int8(). Another parallel work is GLM-130B which uses insights from our work to achieve zero-degradation 8-bit quantization (Zeng et al., 2022). GLM-130B performs full 16-bit precision matrix multiplication with 8-bit weight storage.

## 6 Discussion and Limitations

We have demonstrated for the first time that multi-billion parameter transformers can be quantized to Int8 and used immediately for inference without performance degradation. We achieve this by using our insights from analyzing emergent large magnitude features at scale to develop mixed-precision decomposition to isolate outlier features in a separate 16-bit matrix multiplication. In conjunction with vector-wise quantization that yields our method, LLM.int8(), which we show empirically can recover the full inference performance of models with up to 175B parameters.

The main limitation of our work is that our analysis is solely on the Int8 data type, and we do not study 8-bit floating-point (FP8) data types. Since current GPUs and TPUs do not support this data type, we believe this is best left for future work. However, we also believe many insights from Int8 data types will directly translate to FP8 data types. Another limitation is that we only study models with up to 175B parameters. While we quantize a 175B model to Int8 without performance degradation, additional emergent properties might disrupt our quantization methods at larger scales.

A third limitation is that we do not use Int8 multiplication for the attention function. Since our focus is on reducing the memory footprint and the attention function does not use any parameters, it was not strictly needed. However, an initial exploration of this problem indicated that a solution required additional quantization methods beyond those we developed here, and we leave this for future work.

A final limitation is that we focus on inference but do not study training or finetuning. We provide an initial analysis of Int8 finetuning and training at scale in Appendix E. Int8 training at scale requires complex trade-offs between quantization precision, training speed, and engineering complexity and represents a very difficult problem. We again leave this to future work.

Table 2: Different hardware setups and which methods can be run in 16-bit vs. 8-bit precision. We can see that our 8-bit method makes many models accessible that were not accessible before, in particular, OPT-175B/BLOOM.

Class	Hardware	GPU Memory	Largest Model that can be run	
			8-bit	16-bit
Enterprise	8x A100	80 GB	<b>OPT-175B / BLOOM</b>	<b>OPT-175B / BLOOM</b>
Enterprise	8x A100	40 GB	<b>OPT-175B / BLOOM</b>	OPT-66B
Academic server	8x RTX 3090	24 GB	<b>OPT-175B / BLOOM</b>	OPT-66B
Academic desktop	4x RTX 3090	24 GB	<b>OPT-66B</b>	OPT-30B
Paid Cloud	Colab Pro	15 GB	<b>OPT-13B</b>	GPT-J-6B
Free Cloud	Colab	12 GB	<b>T0/T5-11B</b>	GPT-2 1.3B

## 7 Broader Impacts

The main impact of our work is enabling access to large models that previously could not fit into GPU memory. This enables research and applications which were not possible before due to limited GPU memory, in particular for researchers with the least resources. See Table 3 for model/GPU combinations which are now accessible without performance degradation. However, our work also enables resource-rich organizations with many GPUs to serve more models on the same number of GPUs, which might increase the disparities between resource-rich and poor organizations.

In particular, we believe that the public release of large pretrained models, for example, the recent Open Pretrained Transformers (OPT) (Zhang et al., 2022), along with our new Int8 inference for zero- and few-shot prompting, will enable new research for academic institutions that was not possible before due to resource constraints. The widespread accessibility of such large-scale models will likely have both beneficial and detrimental effects on society that are difficult to predict.

**Acknowledgments** We thank Ofir Press, Gabriel Ilharco, Daniel Jiang, Mitchell Wortsman, Ari Holtzman, Mitchell Gordon for their feedback on drafts of this work. We thank JustHeuristic (Yozh) and Titus von Kölner for help with Hugging Face Transformers integration.

## References

- Artetxe, M., Bhosale, S., Goyal, N., Mihaylov, T., Ott, M., Shleifer, S., Lin, X. V., Du, J., Iyer, S., Pasunuru, R., et al. (2021). Efficient large scale language modeling with mixtures of experts. *arXiv preprint arXiv:2112.10684*.
- Bai, H., Zhang, W., Hou, L., Shang, L., Jin, J., Jiang, X., Liu, Q., Lyu, M. R., and King, I. (2021). Binarybert: Pushing the limit of bert quantization. *ArXiv*, abs/2012.15701.
- Bondarenko, Y., Nagel, M., and Blankevoort, T. (2021). Understanding and overcoming the challenges of efficient transformer quantization. *arXiv preprint arXiv:2109.12948*.
- Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.
- Cambier, L., Bhiwandiwalla, A., Gong, T., Elibol, O. H., Nekuii, M., and Tang, H. (2020). Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.
- Chen, J., Gai, Y., Yao, Z., Mahoney, M. W., and Gonzalez, J. E. (2020). A statistical framework for low-bitwidth training of deep neural networks. *Advances in Neural Information Processing Systems*, 33:883–894.
- Choi, J., Venkataramani, S., Srinivasan, V., Gopalakrishnan, K., Wang, Z., and Chuang, P. (2019). Accurate and efficient 2-bit quantized neural networks. In Talwalkar, A., Smith, V., and Zaharia, M., editors, *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org.
- Courbariaux, M. and Bengio, Y. (2016). Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830.
- Courbariaux, M., Bengio, Y., and David, J. (2015). Binaryconnect: Training deep neural networks with binary weights during propagations. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 3123–3131.
- Courbariaux, M., Bengio, Y., and David, J.-P. (2014). Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*.
- Dettmers, T., Lewis, M., Shleifer, S., and Zettlemoyer, L. (2022). 8-bit optimizers via block-wise quantization. *9th International Conference on Learning Representations, ICLR*.
- Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.
- Dong, Z., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. (2019). Hawq: Hessian aware quantization of neural networks with mixed-precision. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 293–302.
- Esser, S. K., McKinstry, J. L., Bablani, D., Appuswamy, R., and Modha, D. S. (2019). Learned step size quantization. *arXiv preprint arXiv:1902.08153*.
- Fan, A., Stock, P., Graham, B., Grave, E., Gribonval, R., Jegou, H., and Joulin, A. (2020). Training with quantization noise for extreme model compression. *arXiv preprint arXiv:2004.07320*.
- Gao, J., He, D., Tan, X., Qin, T., Wang, L., and Liu, T.-Y. (2019). Representation degeneration problem in training natural language generation models. *arXiv preprint arXiv:1907.12009*.
- Gao, L., Tow, J., Biderman, S., Black, S., DiPofi, A., Foster, C., Golding, L., Hsu, J., McDonell, K., Muennighoff, N., Phang, J., Reynolds, L., Tang, E., Thite, A., Wang, B., Wang, K., and Zou, A. (2021). A framework for few-shot language model evaluation.

- Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., and Keutzer, K. (2021). A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*.
- Gokaslan, A. and Cohen, V. (2019). Openwebtext corpus. *urlhttp://Skylion007. github. io/OpenWebTextCorpus*.
- Gong, R., Liu, X., Jiang, S., Li, T., Hu, P., Lin, J., Yu, F., and Yan, J. (2019). Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pages 4851–4860. IEEE.
- Henighan, T., Kaplan, J., Katz, M., Chen, M., Hesse, C., Jackson, J., Jun, H., Brown, T. B., Dhariwal, P., Gray, S., et al. (2020). Scaling laws for autoregressive generative modeling. *arXiv preprint arXiv:2010.14701*.
- Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., et al. (2022). Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.
- Ilharco, G., Ilharco, C., Turc, I., Dettmers, T., Ferreira, F., and Lee, K. (2020). High performance natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts*, pages 24–27, Online. Association for Computational Linguistics.
- Jin, Q., Ren, J., Zhuang, R., Hanumante, S., Li, Z., Chen, Z., Wang, Y., Yang, K., and Tulyakov, S. (2022). F8net: Fixed-point 8-bit only multiplication for network quantization. *arXiv preprint arXiv:2202.05239*.
- Khudia, D., Huang, J., Basu, P., Deng, S., Liu, H., Park, J., and Smelyanskiy, M. (2021). Fbgemm: Enabling high-performance low-precision deep learning inference. *arXiv preprint arXiv:2101.05615*.
- Kovaleva, O., Kulshreshtha, S., Rogers, A., and Rumshisky, A. (2021). Bert busters: Outlier dimensions that disrupt transformers. *arXiv preprint arXiv:2105.06990*.
- Li, R., Wang, Y., Liang, F., Qin, H., Yan, J., and Fan, R. (2019). Fully quantized network for object detection. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 2810–2819. Computer Vision Foundation / IEEE.
- Lin, Y., Li, Y., Liu, T., Xiao, T., Liu, T., and Zhu, J. (2020). Towards fully 8-bit integer inference for the transformer model. *arXiv preprint arXiv:2009.08034*.
- Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.
- Luo, Z., Kulmizev, A., and Mao, X. (2021). Positional artefacts propagate through masked language model embeddings. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5312–5327, Online. Association for Computational Linguistics.
- Macháček, M. and Bojar, O. (2014). Results of the wmt14 metrics shared task. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 293–301.
- Mellempudi, N., Srinivasan, S., Das, D., and Kaul, B. (2019). Mixed precision training with 8-bit floating point. *CoRR*, abs/1905.12334.
- Nagel, S. (2016). Cc-news.
- Ott, M., Edunov, S., Baevski, A., Fan, A., Gross, S., Ng, N., Grangier, D., and Auli, M. (2019). fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038*.
- Ott, M., Edunov, S., Grangier, D., and Auli, M. (2018). Scaling neural machine translation. *arXiv preprint arXiv:1806.00187*.

- Park, G., Park, B., Kwon, S. J., Kim, B., Lee, Y., and Lee, D. (2022). nuqmm: Quantized matmul for efficient inference of large-scale generative language models. *arXiv preprint arXiv:2206.09557*.
- Puccetti, G., Rogers, A., Drozd, A., and Dell’Orletta, F. (2022). Outliers dimensions that disrupt transformers are driven by frequency. *arXiv preprint arXiv:2205.11380*.
- Qin, H., Gong, R., Liu, X., Bai, X., Song, J., and Sebe, N. (2020). Binary neural networks: A survey. *CoRR*, abs/2004.03333.
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.
- Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2019). Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.
- Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). Xnor-net: Imagenet classification using binary convolutional neural networks. In Leibe, B., Matas, J., Sebe, N., and Welling, M., editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, volume 9908 of *Lecture Notes in Computer Science*, pages 525–542. Springer.
- Sennrich, R., Haddow, B., and Birch, A. (2016). Edinburgh neural machine translation systems for wmt 16. *arXiv preprint arXiv:1606.02891*.
- Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. (2018). Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31.
- Shen, S., Dong, Z., Ye, J., Ma, L., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. (2020). Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8815–8821.
- Sun, X., Choi, J., Chen, C., Wang, N., Venkataramani, S., Srinivasan, V., Cui, X., Zhang, W., and Gopalakrishnan, K. (2019). Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d’Alché-Buc, F., Fox, E. B., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 4901–4910.
- Timkey, W. and van Schijndel, M. (2021). All bark and no bite: Rogue dimensions in transformer language models obscure representational quality. *arXiv preprint arXiv:2109.04404*.
- Trinh, T. H. and Le, Q. V. (2018). A simple method for commonsense reasoning. *arXiv preprint arXiv:1806.02847*.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *arXiv preprint arXiv:1706.03762*.
- Wang, N., Choi, J., Brand, D., Chen, C., and Gopalakrishnan, K. (2018). Training deep neural networks with 8-bit floating point numbers. In Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 7686–7695.
- Wei, X., Zhang, Y., Zhang, X., Gong, R., Zhang, S., Zhang, Q., Yu, F., and Liu, X. (2022). Outlier suppression: Pushing the limit of low-bit transformer language models. *arXiv preprint arXiv:2209.13325*.
- Wenzek, G., Lachaux, M.-A., Conneau, A., Chaudhary, V., Guzmán, F., Joulin, A., and Grave, E. (2020). CCNet: Extracting high quality monolingual datasets from web crawl data. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 4003–4012, Marseille, France. European Language Resources Association.

- Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. (2019). Huggingface’s transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.
- Wu, H., Judd, P., Zhang, X., Isaev, M., and Micikevicius, P. (2020). Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602*.
- Yao, Z., Aminabadi, R. Y., Zhang, M., Wu, X., Li, C., and He, Y. (2022). Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *arXiv preprint arXiv:2206.01861*.
- Yao, Z., Dong, Z., Zheng, Z., Gholami, A., Yu, J., Tan, E., Wang, L., Huang, Q., Wang, Y., Mahoney, M., et al. (2021). Hawq-v3: Dyadic neural network quantization. In *International Conference on Machine Learning*, pages 11875–11886. PMLR.
- Zafirir, O., Boudoukh, G., Izsak, P., and Wasserblat, M. (2019). Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 36–39. IEEE.
- Zeng, A., Liu, X., Du, Z., Wang, Z., Lai, H., Ding, M., Yang, Z., Xu, Y., Zheng, W., Xia, X., et al. (2022). Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414*.
- Zhang, D., Yang, J., Ye, D., and Hua, G. (2018). Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382.
- Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. (2022). Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*.
- Zhang, W., Hou, L., Yin, Y., Shang, L., Chen, X., Jiang, X., and Liu, Q. (2020). Ternarybert: Distillation-aware ultra-low bit bert. In *EMNLP*.
- Zhao, C., Hua, T., Shen, Y., Lou, Q., and Jin, H. (2021). Automatic mixed-precision quantization search of bert. *arXiv preprint arXiv:2112.14938*.
- Zhu, C., Han, S., Mao, H., and Dally, W. J. (2017). Trained ternary quantization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.
- Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., and Fidler, S. (2015). Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27.

## Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default [TODO] to [Yes], [No], or [N/A]. You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] See Section ??.
- Did you include the license to the code and datasets? [No] The code and the data are proprietary.
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...

- (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? **[Yes]**
  - (b) Did you describe the limitations of your work? **[Yes]** See the limitation section
  - (c) Did you discuss any potential negative societal impacts of your work? **[Yes]** See the Broader Impacts section
  - (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? **[Yes]** Yes, we believe our work conforms to these guidelines.
2. If you are including theoretical results...
- (a) Did you state the full set of assumptions of all theoretical results? **[N/A]**
  - (b) Did you include complete proofs of all theoretical results? **[N/A]**
3. If you ran experiments...
- (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? **[Yes]** We will include our code in the supplemental material.
  - (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? **[Yes]** See the experimental setup section
  - (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? **[No]** Our experiments are deterministic for each model. Instead of running the same model multiple times, we run multiple models at different scales. We are unable to compute error bars for these experiments.
  - (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? **[Yes]** See the experimental setup section
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
- (a) If your work uses existing assets, did you cite the creators? **[Yes]** See experimental setup section
  - (b) Did you mention the license of the assets? **[No]** The license is permissible for all the assets that we use. The individual licenses can easily be looked up.
  - (c) Did you include any new assets either in the supplemental material or as a URL? **[N/A]** We only use existing datasets.
  - (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? **[N/A]**
  - (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? **[N/A]**
5. If you used crowdsourcing or conducted research with human subjects...
- (a) Did you include the full text of instructions given to participants and screenshots, if applicable? **[N/A]**
  - (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? **[N/A]**
  - (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? **[N/A]**

## A Memory usage compared to 16-bit precision

Table 3 compares the memory footprint of 16-bit inference and LLM.int8() for different open source models. We can see, that LLM.int8() allows to run the largest open source models OPT-175B and BLOOM-176B on a single node equipped with consumer-grade GPUs.

Table 3: Different hardware setups and which methods can be run in 16-bit vs. 8-bit precision. We can see that our 8-bit method makes many models accessible that were not accessible before, in particular, OPT-175B/BLOOM.

Class	Hardware	GPU Memory	Largest Model that can be run	
			8-bit	16-bit
Enterprise	8x A100	80 GB	<b>OPT-175B / BLOOM</b>	<b>OPT-175B / BLOOM</b>
Enterprise	8x A100	40 GB	<b>OPT-175B / BLOOM</b>	OPT-66B
Academic server	8x RTX 3090	24 GB	<b>OPT-175B / BLOOM</b>	OPT-66B
Academic desktop	4x RTX 3090	24 GB	<b>OPT-66B</b>	OPT-30B
Paid Cloud	Colab Pro	15 GB	<b>OPT-13B</b>	GPT-J-6B
Free Cloud	Colab	12 GB	<b>T0/T5-11B</b>	GPT-2 1.3B

## B Additional Related Work

**Quantization of Transformers with fewer than 1B Parameters** Quantization of transformers has been focused on sub-billion parameter masked language model (MLMs), including BERT (Devlin et al., 2018) and RoBERTa (Liu et al., 2019). Versions of 8-bit BERT/RoBERTa include Q8BERT (Zafirir et al., 2019), QBERT (Shen et al., 2020), product quantization with quantization noise (Fan et al., 2020), TernaryBERT (Zhang et al., 2020), and BinaryBERT (Bai et al., 2021). Work by Zhao et al. (2021) performs both quantization and pruning. All these models require either quantization-aware finetuning or post-training quantization to make the model usable in low-precision. In contrast with our methods, the model can be used directly without performance degradation.

If one views matrix multiplication as 1x1 convolution, vector-wise quantization is equivalent to channel-wise quantization for convolution combined with row quantization (Khudia et al., 2021). For matrix multiplication, this was used by Wu et al. (2020) for BERT-sized transformers (350M parameters), while we are the first to study vector-wise quantization for autoregressive and large-scale models. The only other work that we are aware of that quantizes transformers other than BERT is Chen et al. (2020), which uses post-training quantization with zeropoint quantization in the forward pass and zeropoint-row-wise quantization in the backward pass. However, this work is still for sub-billion parameter transformers. We compare with both zeropoint and row-wise quantization in our evaluations and do not require post-training quantization.

**Low-bitwidth and Convolutional Network Quantization** Work that uses less than 8-bits for data types is usually for convolutional networks (CNNs) to reduce their memory footprint and increase inference speed for mobile devices while minimizing model degradation. Methods for different bit-widths have been studied: 1-bit methods (Courbariaux and Bengio, 2016; Rastegari et al., 2016; Courbariaux et al., 2015), 2 to 3-bit (Zhu et al., 2017; Choi et al., 2019), 4-bits (Li et al., 2019), more bits (Courbariaux et al., 2014), or a variable amount of bits (Gong et al., 2019). For additional related work, please see the survey of Qin et al. (2020). While we believe that lower than 8-bit width with some performance degradation is possible for billion-scale transformers, we focus on 8-bit transformers that *do not* degrade performance and that can benefit from commonly used GPUs that accelerates inference through Int8 tensor cores.

Another line of work that focuses on convolutional network quantization is to learn adjustments to the quantization procedure to improve quantization errors. For example, using Hessian information (Dong et al., 2019), step-size quantization (Esser et al., 2019), soft quantization (Gong et al., 2019), mixed-precision via linear programming optimization (Yao et al., 2021), and other learned quantization methods (Zhang et al., 2018; Gholami et al., 2021).

Table 4: Summary statistics of outliers with a magnitude of at least 6 that occur in at least 25% of all layers and at least 6% of all sequence dimensions. We can see that the lower the C4 validation perplexity, the more outliers are present. Outliers are usually one-sided, and their quartiles with maximum range show that the outlier magnitude is 3-20x larger than the largest magnitude of other feature dimensions, which usually have a range of [-3.5, 3.5]. With increasing scale, outliers become more and more common in all layers of the transformer, and they occur in almost all sequence dimensions. A phase transition occurs at 6.7B parameters when the same outlier occurs in all layers in the same feature dimension for about 75% of all sequence dimensions (SDim). Despite only making up about 0.1% of all features, the outliers are essential for large softmax probabilities. The mean top-1 softmax probability shrinks by about 20% if outliers are removed. Because the outliers have mostly asymmetric distributions across the sequence dimension  $s$ , these outlier dimensions disrupt symmetric softmax quantization and favor asymmetric zeropoint quantization. This explains the results in our validation perplexity analysis. These observations appear to be universal as they occur for models trained in different software frameworks (fairseq, OpenAI, Tensorflow-mesh), and they occur in different inference frameworks (fairseq, Hugging Face Transformers). These outliers also appear robust to slight variations of the transformer architecture (rotary embeddings, embedding norm, residual scaling, different initializations).

Model	PPL $\downarrow$	Params	Outliers		Frequency			Top-1 softmax p	
			Count	1-sided	Layers	SDims	Quartiles	w/ Outlier	No Outlier
GPT2	33.5	117M	1	1	25%	6%	(-8, -7, -6)	45%	19%
GPT2	26.0	345M	2	1	29%	18%	(6, 7, 8)	45%	19%
FSEQ	25.7	125M	2	2	25%	22%	(-40, -23, -11)	32%	24%
GPT2	22.6	762M	2	0	31%	16%	(-9, -6, 9)	41%	18%
GPT2	21.0	1.5B	2	1	41%	35%	(-11, -9, -7)	41%	25%
FSEQ	15.9	1.3B	4	3	64%	47%	(-33, -21, -11)	39%	15%
FSEQ	14.4	2.7B	5	5	52%	18%	(-25, -16, -9)	45%	13%
GPT-J	13.8	6.0B	6	6	62%	28%	(-21, -17, -14)	55%	10%
FSEQ	13.3	6.7B	6	6	100%	75%	(-44, -40, -35)	35%	13%
FSEQ	12.5	13B	7	6	100%	73%	(-63, -58, -45)	37%	16%

## C Detailed Outlier Feature Data

Table 4 provides tabulated data from our outlier feature analysis. We provide the quartiles of the most common outlier in each transformer and the number of outliers that are one-sided, that is, which have asymmetric distributions which do not cross zero.

## D Inference Speedups and Slowdowns

### D.1 Matrix Multiplication benchmarks

While our work focuses on memory efficiency to make models accessible, Int8 methods are also often used to accelerate inference. We find that the quantization and decomposition overhead is significant, and Int8 matrix multiplication itself only yields an advantage if the entire GPU is well saturated, which is only true for large matrix multiplication. This occurs only in LLMs with a model dimension of 4096 or larger.

Detailed benchmarks of raw matrix multiplication and quantization overheads are seen in Table 5. We see that raw Int8 matrix multiplication in cuBLASLt begins to be two times faster than cuBLAS at a model size of 5140 (hidden size 20560). If inputs need to be quantized and outputs dequantized – a strict requirement if not the entire transformer is done in Int8 – then the speedups compared to 16-bit is reduced to 1.6x at a model size of 5140. Models with model size 2560 or smaller are slowed down. Adding mixed precision decomposition slows inference further so that only the 13B and 175B models have speedups.

These numbers could be improved significantly with optimized CUDA kernels for the mixed precision decomposition. However, we also see that existing custom CUDA kernels are much faster than when we use default PyTorch and NVIDIA-provided kernels for quantization which slow down all matrix multiplications except for a 175B model.

Table 5: Inference speedups compared to 16-bit matrix multiplication for the first hidden layer in the feed-forward of differently sized GPT-3 transformers. The hidden dimension is 4x the model dimension. The 8-bit without overhead speedups assumes that no quantization or dequantization is performed. Numbers small than 1.0x represent slowdowns. Int8 matrix multiplication speeds up inference only for models with large model and hidden dimensions.

GPT-3 Size Model dimension	Small 768	Medium 1024	Large 1536	XL 2048	2.7B 2560	6.7B 4096	13B 5140	175B 12288
FP16-bit baseline	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
Int8 without overhead	0.99x	1.08x	1.43x	1.61x	1.63x	1.67x	2.13x	2.29x
Absmax PyTorch+NVIDIA	0.25x	0.24x	0.36x	0.45x	0.53x	0.70x	0.96x	1.50x
Vector-wise PyTorch+NVIDIA	0.21x	0.22x	0.33x	0.41x	0.50x	0.65x	0.91x	1.50x
Vector-wise	<b>0.43x</b>	<b>0.49x</b>	<b>0.74x</b>	<b>0.91x</b>	<b>0.94x</b>	<b>1.18x</b>	<b>1.59x</b>	<b>2.00x</b>
LLM.int8() (vector-wise+decomp)	0.14x	0.20x	0.36x	0.51x	0.64x	0.86x	1.22x	1.81x

## D.2 End-to-end benchmarks

Besides matrix multiplication benchmarks, we also test the end-to-end inference speed of BLOOM-176B in Hugging Face. Hugging Face uses an optimized implementation with cached attention values. Since this type of inference is distributed and, as such, communication dependent, we expect the overall speedup and slowdown due to Int8 inference to be smaller since a large part of the overall inference runtime is the fixed communication overhead.

We benchmark vs. 16-bit and try settings that use a larger batch size or fewer GPUs in the case of Int8 inference, since we can fit the larger model on fewer devices. We can see results for our benchmark in Table 6. Overall Int8 inference is slightly slower but close to the millisecond latency per token compared to 16-bit inference.

Table 6: Ablation study on the number of GPUs used to run several types of inferences of BLOOM-176B model. We compare the number of GPUs used by our quantized BLOOM-176B model together with the native BLOOM-176B model. We also report the *per-token* generation speed in milliseconds for different batch sizes. We use our method integrated into transformers(Wolf et al., 2019) powered by accelerate library from HuggingFace to deal with multi-GPU inference. Our method reaches a similar performance to the native model by fitting into fewer GPUs than the native model.

Batch Size	Hardware	1	8	32
bfloat16 baseline	8xA100 80GB	<b>239</b>	<b>32</b>	9.94
LLM.int8()	8xA100 80GB	253	34	10.44
LLM.int8()	4xA100 80GB	246	33	9.40
LLM.int8()	3xA100 80GB	247	33	<b>9.11</b>

## E Training Results

We test Int8 training on a variety of training settings and compare to 32-bit baselines. We test separate settings for running the transformer with 8-bit feed-forward networks with and without 8-bit linear projections in the attention layer, as well at the attention itself in 8-bit and compare against 32-bit performance. We test two tasks (1) language modeling on part of the RoBERTa corpus including Books (Zhu et al., 2015), CC-News (Nagel, 2016), OpenWebText (Gokaslan and Cohen, 2019), and CC-Stories (Trinh and Le, 2018); and (2) neural machine translation (NMT) (Ott et al., 2018) on WMT14+WMT16 (Macháček and Bojar, 2014; Sennrich et al., 2016).

The results are shown in Table 7 and Table 8. We can see that for training, using the attention linear projections with Int8 data types and vector-wise quantization leads to degradation for NMT and for 1.1B language model but not for 209M language modeling. The results improve slightly if mixed-precision decomposition is used but is not sufficient to recover full performance in most cases. These suggests that training with 8-bit FFN layers is straightforward while other layers require

additional techniques or different data types than Int8 to do 8-bit training at scale without performance degradation.

Table 7: Initial results on small and large-scale language modeling. Doing attention in 8-bit severely degrades performance and performance cannot fully recovered with mixed-precision decomposition. While small-scale language models is close to baseline performance for both 8-bit FFN and 8-bit linear projects in the attention layers performance degrades at the large scale.

Params	Is 8-bit				
	FFN	Linear	Attention	Decomp	PPL
209M				0%	16.74
209M	✓			0%	16.77
209M	✓	✓.		0%	16.83
209M	✓	✓		2%	16.78
209M	✓	✓		5%	16.77
209M	✓	✓		10%	16.80
209M	✓	✓	✓	2%	24.33
209M	✓	✓	✓	5%	20.00
209M	✓	✓	✓	10%	19.00
1.1B				0%	9.99
1.1B	✓			0%	9.93
1.1B	✓	✓		0%	10.52
1.1B	✓	✓		1%	10.41

## F Fine-tuning Results

We also test 8-bit finetuning on RoBERTa-large finetuned on GLUE. We run two different setups: (1) we compare with other Int8 methods, and (2) we compare degradation of finetuning with 8-bit FFN layers as well as 8-bit attention projection layers compared to 32-bit. We finetune with 5 random seeds and report median performance.

Table 9 compares with different previous 8-bit methods for finetuning and shows that vector-wise quantization improves on other methods. Table 10 shows the performance of FFN and/or linear attention projections in 8-bit as well as improvements if mixed-precision decomposition is used. We find that 8-bit FFN layers lead to no degradation while 8-bit attention linear projections lead to degradation if not combined with mixed-precision decomposition where at least the top 2% magnitude dimensions are computed in 16-bit instead of 8-bit. These results highlight the critical role of mixed-precision decomposition for finetuning if one wants to not degrade performance.

Table 8: Neural machine translation results for 8-bit FFN and linear attention layers for WMT14+16. Decomp indicates the percentage that is computed in 16-bit instead of 8-bit. The BLEU score is the median of three random seeds.

Is 8-bit			
FFN	Linear	Decomp	BLEU
		0%	28.9
✓		0%	28.8
✓	✓	0%	unstable
✓	✓	2%	28.0
✓	✓	5%	27.6
✓	✓	10%	27.5

Table 9: GLUE finetuning results for quantization methods for the feedforward layer in 8-bit while the rest is in 16-bit. No mixed-precision decomposition is used. We can see that vector-wise quantization improve upon the baselines.

Method	MNLI	QNLI	QQP	RTE	SST-2	MRPC	CoLA	STS-B	Mean
32-bit Baseline	90.4	94.9	92.2	84.5	96.4	90.1	67.4	93.0	88.61
32-bit Replication	90.3	94.8	92.3	85.4	96.6	90.4	68.8	92.0	88.83
Q-BERT (Shen et al., 2020)	87.8	93.0	90.6	84.7	94.8	88.2	65.1	91.1	86.91
Q8BERT (Zafrrir et al., 2019)	85.6	93.0	90.1	84.8	94.7	89.7	65.0	91.1	86.75
PSQ (Chen et al., 2020)	89.9	94.5	92.0	<b>86.8</b>	96.2	90.4	67.5	91.9	88.65
Vector-wise	<b>90.2</b>	<b>94.7</b>	<b>92.3</b>	85.4	<b>96.4</b>	<b>91.0</b>	<b>68.6</b>	<b>91.9</b>	<b>88.81</b>

Table 10: Breakdown for 8-bit feedforward network (FFN) and linear attention layers for GLUE. Scores are median of 5 random seeds. Decomp indicates the percentage that is decomposed into 16-bit matrix multiplication. Compared to inference, fine-tuning appears to need a higher decomp percentage if the linear attention layers are also converted to 8-bit.

Is 8-bit											MEAN
FFN	Linear	Decomp	MNLI	QNLI	QQP	RTE	SST-2	MRPC	CoLA	STS-B	MEAN
		0%	90.4	94.9	92.2	84.5	96.4	90.1	67.4	93.0	88.6
✓		0%	90.2	94.7	92.3	85.4	96.4	91.0	68.6	91.9	88.8
✓	✓	0%	90.2	94.4	92.2	84.1	96.2	89.7	63.6	91.6	87.7
✓	✓	1%	90.0	94.6	92.2	83.0	96.2	89.7	65.8	91.8	87.9
✓	✓	2%	90.0	94.5	92.2	85.9	96.7	90.4	68.0	91.9	88.7
✓	✓	3%	90.0	94.6	92.2	86.3	96.4	90.2	68.3	91.8	88.7

# Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism

Mohammad Shoeybi<sup>1,2</sup> Mostofa Patwary<sup>1,2</sup> Raul Puri<sup>1,2</sup> Patrick LeGresley<sup>2</sup> Jared Casper<sup>2</sup>  
Bryan Catanzaro<sup>2</sup>

## Abstract

Recent work in language modeling demonstrates that training large transformer models advances the state of the art in Natural Language Processing applications. However, very large models can be quite difficult to train due to memory constraints. In this work, we present our techniques for training very large transformer models and implement a simple, efficient intra-layer model parallel approach that enables training transformer models with billions of parameters. Our approach does not require a new compiler or library changes, is orthogonal and complimentary to pipeline model parallelism, and can be fully implemented with the insertion of a few communication operations in native PyTorch. We illustrate this approach by converging transformer based models up to 8.3 billion parameters using 512 GPUs. We sustain 15.1 PetaFLOPs across the entire application with 76% scaling efficiency when compared to a strong single GPU baseline that sustains 39 TeraFLOPs, which is 30% of peak FLOPs. To demonstrate that large language models can further advance the state of the art (SOTA), we train an 8.3 billion parameter transformer language model similar to GPT-2 and a 3.9 billion parameter model similar to BERT. We show that careful attention to the placement of layer normalization in BERT-like models is critical to achieving increased performance as the model size grows. Using the GPT-2 model we achieve SOTA results on the WikiText103 (10.8 compared to SOTA perplexity of 15.8) and LAMBADA (66.5% compared to SOTA accuracy of 63.2%) datasets. Our BERT model achieves SOTA results on the RACE dataset (90.9% compared to SOTA accuracy of 89.4%).

## 1. Introduction

Natural Language Processing (NLP) is advancing quickly in part due to an increase in available compute and dataset size. The abundance of compute and data enables training increasingly larger language models via unsupervised pretraining (Devlin et al., 2018; Radford et al., 2019). Empirical evidence indicates that larger language models are dramatically more useful for NLP tasks such as article completion, question answering, and natural language inference (Lan et al., 2019; Raffel et al., 2019). By finetuning these pretrained language models on downstream natural language tasks, one can achieve state of the art results as shown in recent work (Devlin et al., 2018; Peters et al., 2018; Howard & Ruder, 2018; Radford et al., 2018; 2017; Ramachandran et al., 2016; Liu et al., 2019b; Dai et al., 2019; Yang et al., 2019; Liu et al., 2019a; Lan et al., 2019).

As these models become larger, they exceed the memory limit of modern processors, and require additional memory management techniques such as activation checkpointing (Chen et al., 2016). Widely used optimization algorithms such as ADAM require additional memory per parameter to store momentum and other optimizer state, which reduces the size of models that can be effectively trained. Several approaches to model parallelism overcome this limit by partitioning the model such that the weights and their associated optimizer state do not need to reside concurrently on the processor. For example, GPipe (Huang et al., 2018) and Mesh-Tensorflow (Shazeer et al., 2018) provide frameworks for model parallelism of different kinds. However, they require rewriting the model, and rely on custom compilers and frameworks that are still under development.

In this work, we implement a simple and efficient model parallel approach using intra-layer model-parallelism. We exploit the inherent structure in transformer based language models to make a simple model-parallel implementation that trains efficiently in PyTorch, with no custom C++ code or compiler required. This approach is orthogonal to pipeline-based model parallelism as advocated by approaches such as GPipe (Huang et al., 2018).

To demonstrate the scalability of our approach, we establish

<sup>1</sup>Equal contribution <sup>2</sup>NVIDIA. Correspondence to: Mohammad Shoeybi <mshoeybi@nvidia.com>.

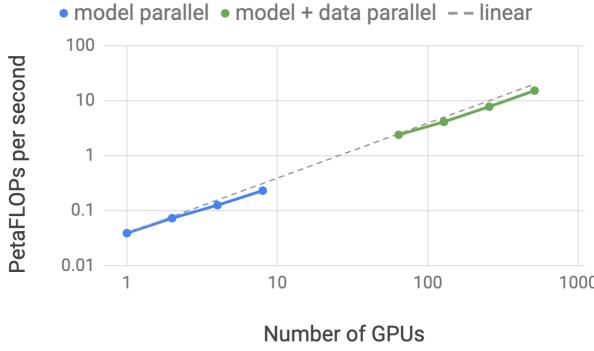


Figure 1. Model (blue) and model+data (green) parallel FLOPS as a function of number of GPUs. Model parallel (blue): up to 8-way model parallel weak scaling with approximately 1 billion parameters per GPU (e.g. 2 billion for 2 GPUs and 4 billion for 4 GPUs). Model+data parallel (green): similar configuration as model parallel combined with 64-way data parallel.

a baseline by training a model of 1.2 billion parameters on a single NVIDIA V100 32GB GPU, that sustains 39 TeraFLOPs. This is 30% of the theoretical peak FLOPS for a single GPU as configured in a DGX-2H server, and is thus a strong baseline. Scaling the model to 8.3 billion parameters on 512 GPUs with 8-way model parallelism, we achieve up to 15.1 PetaFLOPs per second sustained over the entire application. This is 76% scaling efficiency compared to the single GPU case. Figure 1 shows more detailed scaling results.

To analyze the effect of model size scaling on accuracy, we train both left-to-right GPT-2 (Radford et al., 2019) language models as well as BERT (Devlin et al., 2018) bidirectional transformers and evaluate them on several downstream tasks. We show that the existing BERT architecture results in model degradation as the size increases. We overcome this challenge by rearranging the layer normalization and residual connection in the transformer layers and show that with this change, results for the downstream tasks on development sets improve monotonically as the model size increases. In addition, we show that our models achieve test set state of the art (SOTA) results on WikiText103, cloze-style prediction accuracy on LAMBADA, and reading comprehension RACE datasets.

In summary, our contributions are as follows:

- We implement a simple and efficient model parallel approach by making only a few targeted modifications to an existing PyTorch transformer implementation.
- We perform an in-depth empirical analysis of our model and data parallel technique and demonstrate up to 76% scaling efficiency using 512 GPUs.

- We show that careful attention to the placement of layer normalization in BERT-like models is critical to achieving increased accuracies as the model grows.
- We demonstrate that scaling the model size results in improved accuracies for both GPT-2 (studied up to 8.3 billion parameters) and BERT (studied up to 3.9B parameters) models.
- We showcase that our models achieve state of the art results on test sets: perplexity on WikiText103 (10.8 ppl), accuracy on LAMBADA (66.5%), and accuracy on RACE (90.9%).
- We open source our code along with the training and evaluation pipelines at <https://github.com/NVIDIA/Megatron-LM>

## 2. Background and Challenges

### 2.1. Neural Language Model Pretraining

Pretrained language models have become an indispensable part of NLP researchers’ toolkits. Leveraging large corpus pretraining to learn robust neural representations of language is an active area of research that has spanned the past decade. Early examples of pretraining and transferring neural representations of language demonstrated that pre-trained word embedding tables improve downstream task results compared to word embedding tables learned from scratch (Mikolov et al., 2013; Pennington et al., 2014; Turian et al., 2010). Later work advanced research in this area by learning and transferring neural models that capture contextual representations of words (Melamud et al., 2016; McCann et al., 2017; Peters et al., 2018; Radford et al., 2017; 2019). Recent parallel work (Ramachandran et al., 2016; Howard & Ruder, 2018; Radford et al., 2018; Devlin et al., 2018; Liu et al., 2019b; Dai et al., 2019; Yang et al., 2019; Liu et al., 2019a; Lan et al., 2019) further builds upon these ideas by not just transferring the language model to extract contextual word representations, but by also finetuning the language model in an end to end fashion on downstream tasks. Through these works, the state of the art has advanced from transferring just word embedding tables to transferring entire multi-billion parameter language models. This progression of methods has necessitated the need for hardware, systems techniques, and frameworks that are able to operate efficiently at scale and satisfy increasing computational needs. Our work aims to provide the tools necessary to take another step forward in this trend.

### 2.2. Transformer Language Models and Multi-Head Attention

Current work in NLP trends towards using *transformer* models (Vaswani et al., 2017) due to their superior accuracy

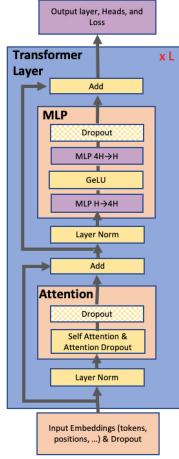


Figure 2. Transformer Architecture. Purple blocks correspond to fully connected layers. Each blue block represents a single transformer layer that is replicated  $N$  times.

and compute efficiency. The original transformer formulation was designed as a machine translation architecture that transforms an input sequence into another output sequence using two parts, an *Encoder* and *Decoder*. However, recent work leveraging transformers for language modeling such as BERT (Devlin et al., 2018) and GPT-2 (Radford et al., 2019) use only the *Encoder* or *Decoder* depending on their needs. This work explores both a decoder architecture, GPT-2, and an encoder architecture, BERT.

Figure 2 shows a schematic diagram of the model we used. We refer the reader to prior work for a detailed description of the model architecture (Vaswani et al., 2017; Devlin et al., 2018; Radford et al., 2019). It is worthwhile to mention that both GPT-2 and BERT use GeLU (Hendrycks & Gimpel, 2016) nonlinearities and layer normalization (Ba et al., 2016) to the input of the multi-head attention and feed forward layers, whereas the original transformer (Vaswani et al., 2017) uses ReLU nonlinearities and applies layer normalization to outputs.

### 2.3. Data and Model Parallelism in Deep Learning

There are two central paradigms for scaling out deep neural network training to numerous hardware accelerators: data parallelism (Valiant, 1990) where a training minibatch is split across multiple workers, and model parallelism in which the memory usage and computation of a model is distributed across multiple workers. By increasing the mini-batch size proportionally to the number of available workers (i.e. *weak scaling*), one observes near linear scaling in training data throughput. However, large batch training introduces complications into the optimization process that can result in reduced accuracy or longer time to convergence, offsetting the benefit of increased training throughput (Keskar et al., 2017). Further research (Goyal et al., 2017; You et al., 2017; 2019) has developed techniques to mitigate

these effects and drive down the training time of large neural networks. To scale out training even further, parallel work (Chen et al., 2016) has combined data parallelism with activation checkpointing: recomputing activations in the backward pass without storing them in the forward pass to reduce memory requirements.

However, these techniques have one fundamental limitation in the problem size they can tackle: the model must fit entirely on one worker. With language models of increasing size and complexity like BERT and GPT-2, neural networks have approached the memory capacity of modern hardware accelerators. One solution to this problem is to employ parameter sharing to reduce the memory footprint of the model (Lan et al., 2019), but this limits the overall capacity of the model. Our approach is to utilize model parallelism to split the model across multiple accelerators. This not only alleviates the memory pressure, but also increases the amount of parallelism independently of the microbatch size.

Within model parallelism, there are two further paradigms: layer-wise pipeline parallelism, and more general distributed tensor computation. In pipeline model parallelism, groups of operations are performed on one device before the outputs are passed to the next device in the pipeline where a different group of operations are performed. Some approaches (Harlap et al., 2018; Chen et al., 2018) use a parameter server (Li et al., 2014) in conjunction with pipeline parallelism. However these suffer from inconsistency issues. The GPipe framework for TensorFlow (Huang et al., 2018) overcomes this inconsistency issue by using synchronous gradient descent. This approach requires additional logic to handle the efficient pipelining of these communication and computation operations, and suffers from pipeline bubbles that reduce efficiency, or changes to the optimizer itself which impact accuracy.

Distributed tensor computation is an orthogonal and more general approach that partitions a tensor operation across multiple devices to accelerate computation or increase model size. FlexFlow (Jia et al., 2018), a deep learning framework orchestrating such parallel computation, provides a method to pick the best parallelization strategy. Recently, Mesh-TensorFlow (Shazeer et al., 2018) introduced a language for specifying a general class of distributed tensor computations in TensorFlow (Abadi et al., 2015). The parallel dimensions are specified in the language by the end user and the resulting graph is compiled with proper collective primitives. We utilize similar insights to those leveraged in Mesh-TensorFlow and exploit parallelism in computing the transformer’s attention heads to parallelize our transformer model. However, rather than implementing a framework and compiler for model parallelism, we make only a few targeted modifications to existing PyTorch transformer implementations. Our approach is simple, does not

require any new compiler or code re-writing, and can be fully implemented by inserting a few simple primitives, as described in the next section.

### 3. Model Parallel Transformers

We take advantage of the structure of transformer networks to create a simple model parallel implementation by adding a few synchronization primitives. A transformer layer consists of a self attention block followed by a two-layer, multi-layer perceptron (MLP) as shown in Figure 2. We introduce model parallelism in both of these blocks separately.

We start by detailing the MLP block. The first part of the block is a GEMM followed by a GeLU nonlinearity:

$$Y = \text{GeLU}(XA) \quad (1)$$

One option to parallelize the GEMM is to split the weight matrix  $A$  along its rows and input  $X$  along its columns as:

$$X = [X_1, X_2], A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}. \quad (2)$$

This partitioning will result in  $Y = \text{GeLU}(X_1A_1 + X_2A_2)$ . Since GeLU is a nonlinear function,  $\text{GeLU}(X_1A_1 + X_2A_2) \neq \text{GeLU}(X_1A_1) + \text{GeLU}(X_2A_2)$  and this approach will require a synchronization point before the GeLU function.

Another option is to split  $A$  along its columns  $A = [A_1, A_2]$ . This partitioning allows the GeLU nonlinearity to be independently applied to the output of each partitioned GEMM:

$$[Y_1, Y_2] = [\text{GeLU}(XA_1), \text{GeLU}(XA_2)] \quad (3)$$

This is advantageous as it removes a synchronization point. Hence, we partition the first GEMM in this column parallel fashion and split the second GEMM along its rows so it takes the output of the GeLU layer directly without requiring any communication as shown in Figure 3a. The output of the second GEMM is then reduced across the GPUs before passing the output to the dropout layer. This approach splits both GEMMs in the MLP block across GPUs and requires only a single all-reduce operation in the forward pass ( $g$  operator) and a single all-reduce in the backward pass ( $f$  operator). These two operators are conjugates of each other and can be implemented in PyTorch with only a few lines of code. As an example, the implementation of the  $f$  operator is provided below:

```
class f(torch.autograd.Function):
    def forward(ctx, x):
        return x
    def backward(ctx, gradient):
        all_reduce(gradient)
        return gradient
```

Code 1. Implementation of  $f$  operator.  $g$  is similar to  $f$  with identity in the backward and all-reduce in the forward functions.

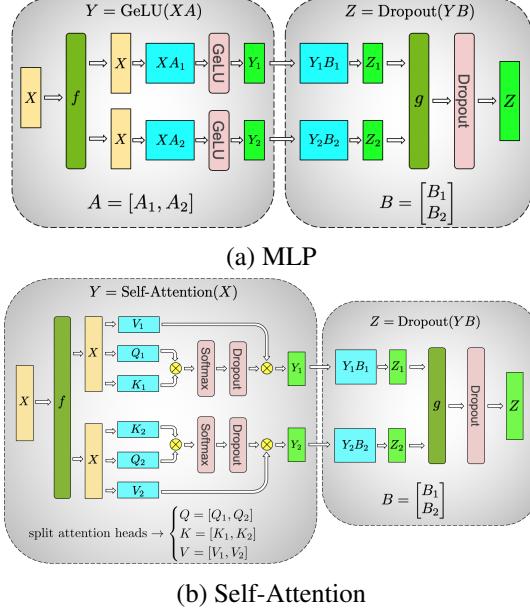


Figure 3. Blocks of Transformer with Model Parallelism.  $f$  and  $g$  are conjugate.  $f$  is an identity operator in the forward pass and all reduce in the backward pass while  $g$  is an all reduce in the forward pass and identity in the backward pass.

As shown in Figure 3b, for the self attention block we exploit inherent parallelism in the multihead attention operation, partitioning the GEMMs associated with key ( $K$ ), query ( $Q$ ), and value ( $V$ ) in a column parallel fashion such that the matrix multiply corresponding to each attention head is done locally on one GPU. This allows us to split per attention head parameters and workload across the GPUs, and doesn't require any immediate communication to complete the self-attention. The subsequent GEMM from the output linear layer (after self attention) is parallelized along its rows and takes the output of the parallel attention layer directly, without requiring communication between the GPUs. This approach for both the MLP and self attention layer fuses groups of two GEMMs, eliminates a synchronization point in between, and results in better scaling. This enables us to perform all GEMMs in a simple transformer layer using only two all-reduces in the forward path and two in the backward path (see Figure 4).

The transformer language model has an output embedding with the dimension of hidden-size ( $H$ ) times vocabulary-size ( $v$ ). Since the vocabulary size is on the order of tens of thousands of tokens for modern language models (for example, GPT-2 used a vocabulary size of 50,257), it is beneficial to parallelize the output embedding GEMM. However, in transformer language models, the output embedding layer shares weights with the input embedding, requiring modifications to both. We parallelize the input embedding weight matrix  $E_{H \times v}$  along the vocabulary dimension  $E = [E_1, E_2]$  (column-wise). Since each partition now only

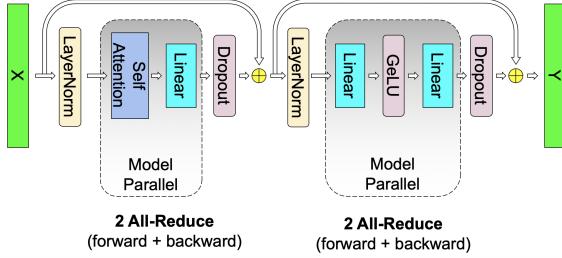


Figure 4. Communication operations in a transformer layer. There are 4 total communication operations in the forward and backward pass of a single model parallel transformer layer.

contains a portion of the embedding table, an all-reduce ( $g$  operator) is required after the input embedding. For the output embedding, one approach is to perform the parallel GEMM  $[Y_1, Y_2] = [X E_1, X E_2]$  to obtain the logits, add an all-gather  $Y = \text{all-gather}([Y_1, Y_2])$ , and send the results to the cross-entropy loss function. However, for this case, the all-gather will communicate  $b \times s \times v$  elements ( $b$  is the batch-size and  $s$  is the sequence length) which is huge due to vocabulary size being large. To reduce the communication size, we fuse the output of the parallel GEMM  $[Y_1, Y_2]$  with the cross entropy loss which reduces the dimension to  $b \times s$ . Communicating scalar losses instead of logits is a huge reduction in communication that improves the efficiency of our model parallel approach.

Much of our model parallel approach can be characterized as techniques aimed at reducing communication and keeping the GPUs compute bound. Rather than having one GPU compute part of the dropout, layer normalization, or residual connections and broadcast the results to other GPUs, we choose to duplicate the computation across GPUs. Specifically, we maintain duplicate copies of layer normalization parameters on each GPU, and take the output of the model parallel region and run dropout and residual connection on these tensors before feeding them as input to the next model parallel regions. To optimize the model we allow each model parallel worker to optimize its own set of parameters. Since all values are either local to or duplicated on a GPU, there is no need for communicating updated parameter values in this formulation.

We present further details about the hybrid model and data parallelism and handling random number generation in Appendix B for reference. In summary, our approach as described above is simple to implement, requiring only a few extra all-reduce operations added to the forward and backward pass. It does not require a compiler, and is orthogonal and complementary to the pipeline model parallelism advocated by approaches such as (Huang et al., 2018).

## 4. Setup

Pretrained language understanding models are central tasks in natural language processing and language understanding. There are several formulations of language modeling. In this work we focus on GPT-2 (Radford et al., 2019), a left-to-right generative transformer based language model, and BERT (Devlin et al., 2018), a bi-directional transformer model based on language model masking. We explain our configurations for these models in the following section and refer to the original papers for more details.

### 4.1. Training Dataset

To collect a large diverse training set with longterm dependencies we aggregate several of the largest language modeling datasets. We create an aggregate dataset consisting of Wikipedia (Devlin et al., 2018), CC-Stories (Trinh & Le, 2018), RealNews (Zellers et al., 2019), and OpenWebtext (Radford et al., 2019). To avoid training set leakage into our downstream tasks we remove the Wikipedia articles present in the WikiText103 test set (Merity et al., 2016). We also remove unnecessary newlines from the CC-Stories corpus introduced by preprocessing artifacts. For BERT models we include BooksCorpus (Zhu et al., 2015) in the training dataset, however, this dataset is excluded for GPT-2 trainings as it overlaps with LAMBADA task.

We combined all the datasets and then filtered out all the documents with content length less than 128 tokens from the aggregated dataset. Since similar content might appear multiple times in the aggregated datasets, we used locality-sensitive hashing (LSH) to deduplicate content with a jaccard similarity greater than 0.7. The resulting aggregate corpus contains 174 GB of deduplicated text.

### 4.2. Training Optimization and Hyperparameters

To train our models efficiently we utilize mixed precision training with dynamic loss scaling to take advantage of the V100’s Tensor Cores (Micikevicius et al., 2017; NVIDIA, 2018). We start by initializing our weights  $W$  with a simple normal distribution  $W \sim \mathcal{N}(0, 0.02)$ . We then scale weights immediately before residual layers by  $\frac{1}{\sqrt{2N}}$  where  $N$  is the number of transformer layers comprised of self attention and MLP blocks. For our optimizer we utilize Adam (Kingma & Ba, 2014) with weight decay (Loshchilov & Hutter, 2019)  $\lambda = 0.01$ . Additionally, we use global gradient norm clipping of 1.0 to improve the stability of training large models. In all cases, a dropout of 0.1 is used. Lastly, to better manage our memory footprint we utilize activation checkpointing (Chen et al., 2016) after every transformer layer.

For GPT-2 models, all training is performed with sequences of 1024 subword units at a batch size of 512 for 300k itera-

tions. Our learning rate of 1.5e-4 utilizes a warmup period of 3k iterations before following a single cycle cosine decay over the remaining 297k iterations. We stop the decay at a minimum learning rate of 1e-5.

For BERT models, we largely follow the training process described in (Lan et al., 2019). We use the original BERT dictionary with vocab size of 30,522. In addition, we replace the next sentence prediction head with sentence order prediction as suggested by (Lan et al., 2019) and use whole word n-gram masking of (Joshi et al., 2019). For all cases, we set the batch size to 1024 and use a learning rate of 1.0e-4 warmed up over 10,000 iterations and decayed linearly over 2 million iterations. Other training parameters are kept the same as (Devlin et al., 2018).

## 5. Experiments

All of our experiments use up to 32 DGX-2H servers (a total of 512 Tesla V100 SXM3 32GB GPUs). Our infrastructure is optimized for multi-node deep learning applications, with 300 GB/sec bandwidth between GPUs inside a server via NVSwitch and 100 GB/sec of interconnect bandwidth between servers using 8 InfiniBand adapters per server.

### 5.1. Scaling Analysis

To test the scalability of our implementation, we consider GPT-2 models with four sets of parameters detailed in Table 1. To have consistent GEMM sizes in the self attention layer, the hidden size per attention head is kept constant at 96 while the number of heads and layers are varied to obtain configurations ranging from 1 billion to 8 billion parameters. The configuration with 1.2 billion parameters fits on a single GPU whereas the 8 billion parameter model requires 8-way model parallelism (8 GPUs). The original vocabulary size was 50,257, however, to have efficient GEMMs for the logit layer, it is beneficial for the per-GPU vocabulary size to be a multiple of 128. Since we study up to 8-way model parallelism, we pad the vocabulary such that it is divisible by  $128 \times 8 = 1024$ , resulting in a padded vocabulary size of 51,200. We study both model and model+data parallel scaling. For the model parallel scaling, a fixed batch size of 8 is used across all configurations. Data parallel scaling is necessary for training many state of the art models which typically use a much larger global batch size. To this end, for the model+data parallel cases we fix the global batch size to 512 for all experiments which corresponds to 64-way data parallelism.

#### 5.1.1. MODEL AND DATA PARALLELISM

Throughout this section, we will showcase weak scaling with respect to the model parameters for both model parallel and model+data parallel cases. Weak scaling is typically

Table 1. Parameters used for scaling studies. Hidden size per attention head is kept constant at 96.

Hidden Size	Attention heads	Number of layers	Number of parameters (billions)	Model parallel GPUs	Model +data parallel GPUs
1536	16	40	1.2	1	64
1920	20	54	2.5	2	128
2304	24	64	4.2	4	256
3072	32	72	8.3	8	512

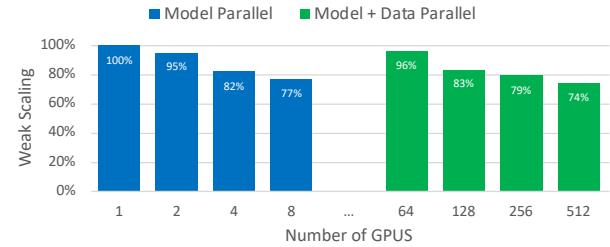


Figure 5. Model and model + data parallel weak scaling efficiency as a function of the number of GPUs.

done by scaling the batch-size, however, this approach does not address training large models that do not fit on a single GPU and it leads to training convergence degradation for large batch sizes. In contrast, here we use weak scaling to train larger models that were not possible otherwise. The baseline for all the scaling numbers is the first configuration (1.2 billion parameters) in Table 1 running on a single GPU. This is a strong baseline as it achieves 39 TeraFLOPS during the overall training process, which is 30% of the theoretical peak FLOPS for a single GPU in a DGX-2H server.

Figure 5 shows scaling values for both model and model+data parallelism. We observe excellent scaling numbers in both settings. For example, the 8.3 billion parameters case with 8-way (8 GPU) model parallelism achieves 77% of linear scaling. Model+data parallelism requires further communication of gradients and as a result the scaling numbers drop slightly. However, even for the largest configuration (8.3 billion parameters) running on 512 GPUs, we achieve 74% scaling relative to linear scaling of the strong single GPU baseline configuration (1.2 billion parameters). Further scaling analysis is provided in Appendix D

### 5.2. Language Modeling Results Using GPT-2

To demonstrate that large language models can further advance the state of the art, we consider training GPT-2 models of the sizes and configurations listed in Table 2. The 355M model is equivalent in size and configuration of BERT-Large model (Devlin et al., 2018). The 2.5B model is bigger than the previous largest GPT-2 model, and the 8.3B model is larger than any left-to-right transformer language model ever trained, to the best of our knowledge. To train and eval-

Table 2. Model configurations used for GPT-2.

Parameter Count	Layers	Hidden Size	Attn Heads	Hidden Size per Head	Total GPUs	Time per Epoch (days)
355M	24	1024	16	64	64	0.86
2.5B	54	1920	20	96	128	2.27
8.3B	72	3072	24	128	512	2.10

Table 3. Zero-shot results. SOTA are from (Khandelwal et al., 2019) for WikiText103 and (Radford et al., 2019) for LAMBADA.

Model	WikiText103 Perplexity ↓	LAMBADA Accuracy ↑
355M	19.31	45.18%
2.5B	12.76	61.73%
8.3B	<b>10.81</b>	<b>66.51%</b>
Previous SOTA	15.79	63.24%

uate our language models we use the procedure described in section 4. Table 2 also lists the time it takes to advance one epoch which is equivalent to 68,507 iterations. For example, for the 8.3B model on 512 GPUs, each epoch takes around two days. Compared to the configurations used for our scaling studies in Table 1, the 2.5B model is the same, the 8.3B model has 24 attention heads instead of 32, and the 355M is much smaller than any seen previously while still using 64 GPUs to train, leading to the much lower time per epoch.

Figure 6 shows validation perplexity as a function of number of iterations. As the model size increases, the validation perplexity decreases and reaches a validation perplexity of 9.27 for the 8.3B model. We report the zero-shot evaluation of the trained models on the LAMBADA and WikiText103 datasets in Table 3. For more details on evaluation methodology, see Appendix E. We observe the trend that increasing model size also leads to lower perplexity on WikiText103 and higher cloze accuracy on LAMBADA. Our 8.3B model achieves state of the art perplexity on the WikiText103 test set at a properly adjusted perplexity of 10.81. At 66.51% accuracy, the 8.3B model similarly surpasses prior cloze accuracy results on the LAMBADA task. We have included samples generated from the 8.3 billion parameters model in the Appendix C. Recently researchers from Microsoft in collaboration with NVIDIA trained a 17 billion parameter GPT-2 model called Turing-NLG (Microsoft, 2020) using Megatron and showed that the accuracies further improve as they scale the model, highlighting the value of larger models.

To ensure we do not train on any data found in our test sets, we calculate the percentage of test set 8-grams that also appear in our training set as done in previous work (Radford et al., 2019). The WikiText103 test set has at most

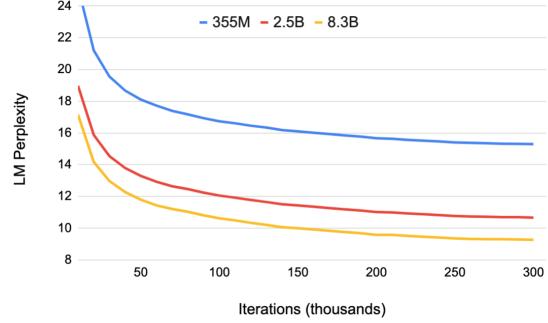


Figure 6. Validation set perplexity. All language models are trained for 300k iterations. Larger language models converge noticeably faster and converge to lower validation perplexities than their smaller counterparts.

Table 4. Model configurations used for BERT.

Parameter Count	Layers	Hidden Size	Attention Heads	Total GPUs
336M	24	1024	16	128
1.3B	24	2048	32	256
3.9B	48	2560	40	512

10.8% overlap and the LAMBADA test set (Paperno et al., 2016) has at most 1.4% overlap. We should note that the WikiText103 test set has already 9.09% overlap with the WikiText103 training set (Radford et al., 2019). As these are consistent with previous work, we are confident that no documents from our test data are inadvertently included in our training data.

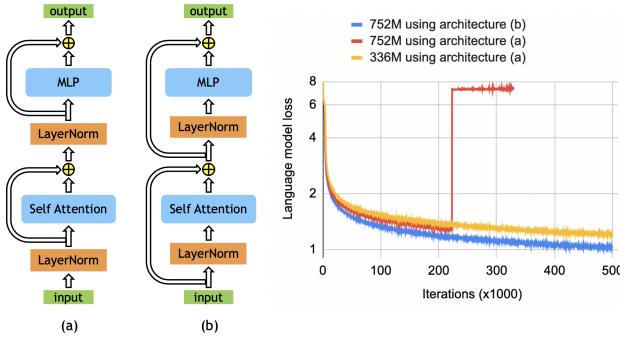
### 5.3. Bi-directional Transformer Results Using BERT

In this section, we apply our methodology to BERT-style transformer models and study the effect of model scaling on several downstream tasks. Prior work (Lan et al., 2019) found that increasing model size beyond BERT-large with 336M parameters results in unexpected model degradation. To address this degradation, the authors of that work (Lan et al., 2019) introduced parameter sharing and showed that that their models scale much better compared to the original BERT model.

We further investigated this behaviour and empirically demonstrated that rearranging the order of the layer normalization and the residual connections as shown in Figure 7 is critical to enable the scaling of the BERT-style models beyond BERT-Large. The architecture (b) in Figure 7 eliminates instabilities observed using the original BERT architecture in (a) and also has a lower training loss. To the best of our knowledge, we are the first to report such a change enables training larger BERT models.

*Table 5.* Development set results for MNLI, QQP, SQuAD 1.1 and SQuAD 2.0 and test set results for RACE. The trained tokens represents consumed tokens during model pretraining (proportional to batch size times number of iterations) normalized by consumed tokens during model pretraining for our 336M model.

Model	trained tokens ratio	MNLI m/mm accuracy (dev set)	QQP accuracy (dev set)	SQuAD 1.1 F1 / EM (dev set)	SQuAD 2.0 F1 / EM (dev set)	RACE m/h accuracy (test set)
RoBERTa (Liu et al., 2019b)	2	90.2 / 90.2	92.2	94.6 / 88.9	89.4 / 86.5	83.2 (86.5 / 81.8)
ALBERT (Lan et al., 2019)	3	90.8	92.2	94.8 / 89.3	90.2 / 87.4	86.5 (89.0 / 85.5)
XLNet (Yang et al., 2019)	2	90.8 / 90.8	92.3	95.1 / 89.7	90.6 / 87.9	85.4 (88.6 / 84.0)
Megatron-336M	1	89.7 / 90.0	92.3	94.2 / 88.0	88.1 / 84.8	83.0 (86.9 / 81.5)
Megatron-1.3B	1	90.9 / 91.0	92.6	94.9 / 89.1	90.2 / 87.1	87.3 (90.4 / 86.1)
Megatron-3.9B	1	<b>91.4 / 91.4</b>	<b>92.7</b>	<b>95.5 / 90.0</b>	<b>91.2 / 88.5</b>	<b>89.5 (91.8 / 88.6)</b>
ALBERT ensemble (Lan et al., 2019)				95.5 / 90.1	91.4 / 88.9	89.4 (91.2 / 88.6)
Megatron-3.9B ensemble				<b>95.8 / 90.5</b>	<b>91.7 / 89.0</b>	<b>90.9 (93.1 / 90.0)</b>



*Figure 7.* Training loss for BERT model using the original architecture (a) and the rearranged architecture (b). Left figure shows the training loss for 336M and 752M BERT model. While the original architecture performs well on the 336M model, the modifications in (b) enable stable training with lower training loss.

Using the architecture change in Figure 7(b), we consider three different cases as detailed in Table 4. The 336M model has the same size as BERT-large. The 1.3B is the same as the BERT-xlarge configuration that was previously shown to get worse results than the 336M BERT-large model (Lan et al., 2019). We further scale the BERT model using both larger hidden size as well as more layers to arrive at the 3.9B parameter case. In all cases, the hidden size per attention head is kept constant at 64. 336M and 1.3B models are trained for 2 million iterations while the 3.9B model is trained for 1.5 million iterations and is still training.

On a 3% held-out set, 336M, 1.3B, and 3.9B models achieve validation set perplexity of 1.58, 1.30, and 1.16, respectively, a monotonic decrease with the model size. We finetune the trained models on several downstream tasks including MNLI and QQP from the GLUE benchmark (Wang et al., 2019), SQuAD 1.1 and SQuAD 2.0 from the Stanford Question answering dataset (Rajpurkar et al., 2016; 2018), and the reading comprehension RACE dataset (Lai et al., 2017). For finetuning, we follow the same procedure as (Liu et al., 2019b). We first perform hyperparameter tuning on batch

size and learning rate. Once we obtain the best values, we report the median development set results over 5 different random seeds for initialization. The hyperparameters used for each model and task are provided in the Appendix A. Table 5 shows the development set results for MNLI, QQP, SQuAD 1.1, and SQuAD 2.0 and test set results for RACE. For the test set results of RACE, we first use the development set to find the checkpoint that gives us the median score on the 5 random seeds and we report the results from that checkpoint on the test set. We also report 5-way ensemble results for the development set of SQuAD and test set of RACE. From Table 5 we observe that (a) as the model size increases, the downstream task performance improves in all cases, (b) our 3.9B model establishes state of the art results on the development set compared to other BERT based models, and (c) our 3.9B model achieves both single model as well as ensembled SOTA results on RACE test set.

## 6. Conclusion and Future Work

In this work, we successfully surpassed the limitations posed by traditional single-GPU-per-model training by implementing model parallelism with only a few modifications to the existing PyTorch transformer implementations. We efficiently trained transformer based models up to 8.3 billion parameter on 512 NVIDIA V100 GPUs with 8-way model parallelism and achieved up to 15.1 PetaFLOPs sustained over the entire application. We also showed that for BERT models, careful attention to the placement of layer normalization in BERT-like models is critical to achieving increased accuracies as the model size increases. We study the effect of model size on down-stream task accuracy and achieve far superior results on downstream tasks and establish new SOTA for WikiText103, LAMBADA, and RACE datasets. Finally, we open sourced our code to enable future work leveraging model parallel transformers.

There are several directions for future work. Continuing to increase the scale of pretraining is a promising line of

investigation that will further test existing deep learning hardware and software. To realize this, improvements in the efficiency and memory footprint of optimizers will be needed. In addition, training a model with more than 16 billion parameters will demand more memory than is available within 16 GPUs of a DGX-2H box. For such models, a hybrid intra-layer and inter-layer model parallelism along with inter-node model parallelism would be more suitable. Three other directions of investigation include (a) pretraining different model families (XLNet, T5), (b) evaluating performance of large models across more difficult and diverse downstream tasks (e.g. Generative Question Answering, Summarization, and Conversation), and (c) using knowledge distillation to train small student models from these large pretrained teacher models.

## References

- Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL <http://tensorflow.org/>. Software available from tensorflow.org.
- Ba, J. L., Kiros, J. R., and Hinton, G. E. LayerNorm. *CoRR*, abs/1607.06450, 2016. URL <http://arxiv.org/abs/1607.06450>.
- Chen, C.-C., Yang, C.-L., and Cheng, H.-Y. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv:1809.02839*, 2018.
- Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016. URL <http://arxiv.org/abs/1604.06174>.
- Dai, Z., Yang, Z., Yang, Y., Carbonell, J. G., Le, Q. V., and Salakhutdinov, R. Transformer-xl: Attentive language models beyond a fixed-length context. *CoRR*, abs/1901.02860, 2019. URL <http://arxiv.org/abs/1901.02860>.
- Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.
- Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.
- Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv:1806.03377*, 2018.
- Hendrycks, D. and Gimpel, K. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016. URL <http://arxiv.org/abs/1606.08415>.
- Howard, J. and Ruder, S. Fine-tuned language models for text classification. *CoRR*, abs/1801.06146, 2018.
- Huang, Y., Cheng, Y., Chen, D., Lee, H., Ngiam, J., Le, Q. V., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *CoRR*, abs/1811.06965, 2018. URL <http://arxiv.org/abs/1811.06965>.
- Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *arXiv:1807.05358*, 2018.
- Joshi, M., Chen, D., Liu, Y., Weld, D. S., Zettlemoyer, L., and Levy, O. Spanbert: Improving pre-training by representing and predicting spans. *arXiv:1907.10529*, 2019.
- Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On large- batch training for deep learning: Generalization gap and sharp minima. *ICLR*, 2017.
- Khandelwal, U., Levy, O., Jurafsky, D., Zettlemoyer, L., and Lewis, M. Generalization through memorization: Nearest neighbor language models. *arXiv:1911.00172*, 2019.
- Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- Lai, G., Xie, Q., Liu, H., Yang, Y., and Hovy, E. Race: Large-scale reading comprehension dataset from examinations. *arXiv:1704.04683*, 2017.
- Lan, Z., Chen, M., Goodman, S., Gimpel, K., and Soricut, P. S. R. Albert: A lite bert for self-supervised learning of language representations. *arXiv:1909.11942*, 2019.
- Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server, 2014.
- Liu, X., He, P., Chen, W., and Gao, J. Multi-task deep neural networks for natural language understanding. *CoRR*, abs/1901.11504, 2019a. URL <http://arxiv.org/abs/1901.11504>.

- Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019b. URL <http://arxiv.org/abs/1907.11692>.
- Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=Bkg6RiCqY7>.
- McCann, B., Bradbury, J., Xiong, C., and Socher, R. Learned in translation: Contextualized word vectors. *CoRR*, abs/1708.00107, 2017.
- Melamud, O., Goldberger, J., and Dagan, I. context2vec: Learning generic context embedding with bidirectional lstm. In *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, pp. 51–61, 01 2016.
- Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *CoRR*, abs/1609.07843, 2016. URL <http://arxiv.org/abs/1609.07843>.
- Micikevicius, P., Narang, S., Alben, J., Diamos, G. F., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training. *CoRR*, abs/1710.03740, 2017.
- Microsoft. Turing-nlg: A 17-billion-parameter language model by microsoft, 2020. URL <https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/>.
- Mikolov, T., Deoras, A., Kombrink, S., Burget, L., and Černocký, J. Empirical evaluation and combination of advanced language modeling techniques. In *Twelfth Annual Conference of the International Speech Communication Association*, 2011.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.
- NVIDIA. Mixed precision training: Choosing a scaling factor, 2018. URL <https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html#scalefactor>.
- Paperno, D., Kruszewski, G., Lazaridou, A., Pham, Q. N., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., and Fernández, R. The LAMBADA dataset: Word prediction requiring a broad discourse context. *CoRR*, abs/1606.06031, 2016. URL <http://arxiv.org/abs/1606.06031>.
- Pennington, J., Socher, R., and Manning, C. D. Glove: Global vectors for word representation, 2014. URL <https://www.aclweb.org/anthology/D14-1162>.
- Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. Deep contextualized word representations. *CoRR*, abs/1802.05365, 2018. URL <http://arxiv.org/abs/1802.05365>.
- Radford, A., Józefowicz, R., and Sutskever, I. Learning to generate reviews and discovering sentiment. *CoRR*, abs/1704.01444, 2017.
- Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding by generative pre-training, 2018. URL <https://blog.openai.com/language-unsupervised/>.
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Better language models and their implications, 2019. URL <https://openai.com/blog/better-language-models/>.
- Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv:1910.10683*, 2019.
- Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. Squad: 100,000+ questions for machine comprehension of text. *EMNLP*, 2016.
- Rajpurkar, P., Jia, R., and Liang, P. Know what you dont know: Unanswerable questions for squad. *ACL*, 2018.
- Ramachandran, P., Liu, P. J., and Le, Q. V. Unsupervised pretraining for sequence to sequence learning. *CoRR*, abs/1611.02683, 2016. URL <http://arxiv.org/abs/1611.02683>.
- Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., Sepassi, R., and Hechtman, B. Mesh-TensorFlow: Deep learning for supercomputers. In *Neural Information Processing Systems*, 2018.
- Trinh, T. H. and Le, Q. V. A simple method for common-sense reasoning. *CoRR*, abs/1806.02847, 2018. URL <http://arxiv.org/abs/1806.02847>.
- Turian, J., Ratinov, L., and Bengio, Y. Word representations: A simple and general method for semi-supervised learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, ACL '10*, pp. 384–394, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

Valiant, L. G. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. Glue: A multi-task benchmark and analysis platform for natural language understanding. *ICLR*, 2019.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J. G., Salakhutdinov, R., and Le, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019. URL <http://arxiv.org/abs/1906.08237>.

You, Y., Gitman, I., and Ginsburg, B. Large batch training of convolutional networks. *arXiv:1708.03888*, 2017.

You, Y., Li, J., Reddi, S., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., and Hsieh, C.-J. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv:1904.00962*, 2019.

Zellers, R., Holtzman, A., Rashkin, H., Bisk, Y., Farhadi, A., Roesner, F., and Choi, Y. Defending against neural fake news. *CoRR*, abs/1905.12616, 2019. URL <http://arxiv.org/abs/1905.12616>.

Zhu, Y., Kiros, R., Zemel, R. S., Salakhutdinov, R., Urtasun, R., Torralba, A., and Fidler, S. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. *CoRR*, abs/1506.06724, 2015.

## A. BERT Finetuning Hyperparameters

Table 6 presents the hyperparameters used for each model and task during finetuning.

## B. Model Parallel Supplementary Material

In this section, we present further details about the hybrid model and data parallelism and handling random number generation.

### B.1. Hybrid Model and Data Parallelism

Model parallelism is orthogonal to data parallelism, and so we can use both simultaneously to train large models in a reasonable amount of time. Figure 8 shows a grouping of GPUs for hybrid model and data parallelism. Two or more GPUs within the same server form model parallel groups (for example GPUs 1 to 8 in Figure 8), and contain one

Table 6. Hyperparameters for finetuning BERT model on downstream tasks.

Task	Model	Batch size	Learning rate	Training epochs
MNLI	336M	128	1e-5	10
	1.3B			
	3.8B			
QQP	336M	128	5e-5	12
	1.3B		3e-5	
	3.8B	256	4e-5	
SQuAD 1.1	336M	64	3e-5	2
	1.3B		3e-5	
	3.8B	48	1e-5	
SQuAD 2.0	336M	48	3e-5	2
	1.3B		3e-5	
	3.8B	48	1e-5	
RACE	336M	32	2e-5	3
	1.3B		1e-5	
	3.8B	32	2e-5	

instance of the model distributed across these GPUs. The remaining GPUs, which could be within the same server but more typically are located in other servers, run additional model parallel groups. GPUs with the same position in each of the model parallel groups (for example GPUs 1, 9, ..., 505 in Figure 8) form data parallel groups so that all GPUs within a data parallel group hold the same model parameters. During back propagation we run multiple gradient all-reduce operations in parallel to reduce weight gradients within each distinct data parallel group. The total number of required GPUs is the product of the number of model and data parallel groups. For example, for the 8.3 billion parameter model we use 8 GPUs per model parallel group and 64-way data parallelism, for a total of 512 GPUs. All communication is implemented in PyTorch by Python calls to NCCL. GPUs within each model parallel group perform all-reduces amongst all GPUs within the group. For data parallelism, each of the all-reduce operations takes place with one of the GPUs from each model parallel group.

### B.2. Model Parallel Random Number Generation

Techniques that utilize random number generation, such as dropout, are a staple of modern deep learning training. Transformers have dropout layers outside the model parallel regions before residual connections and within model parallel regions in the self attention block. Because some dropout layers are in a model parallel region, while others are not, we need to treat random number generation carefully to ensure dropout works correctly. To synchronize residual connection dropout across model parallel workers we seed the random number generators at the beginning of training with the same seed. This results in identical dropout patterns across all model parallel workers. However, dropout within a model parallel region should result in different random

patterns for each worker to achieve randomness across the entire operation. To achieve this we maintain a separate random number generator for dropout within model parallel regions. This random number generator is uniquely seeded for each model parallel worker.

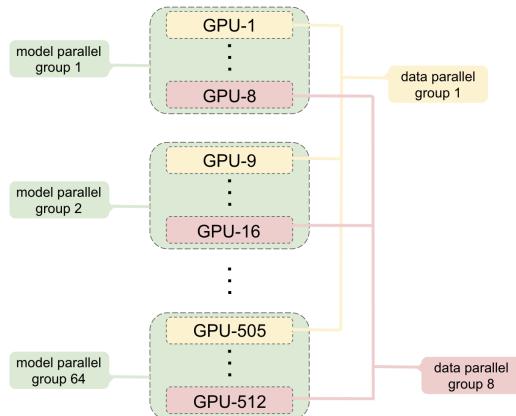


Figure 8. Grouping of GPUs for hybrid model and data parallelism with 8-way model parallel and 64-way data parallel.

**Context:** There is a major force uniting Americas fiercely partisan politicians: big technology companies. Democrats and Republicans at the federal and state levels are coming together to scrutinize the power of the Silicon Valley giants and, potentially, to rein them in. Letitia James, the Democratic attorney general of New York, announced on Friday that attorneys general in eight states four Democrats and four Republicans and the District of Columbia had begun an antitrust investigation of Facebook.

**Megatron-LM:** The probe, which Ms. James said was likely to result in a lawsuit, is the latest in a string of high-profile regulatory actions, in state and federal courts, that challenge the market dominance of tech companies. The Justice Department is also investigating Facebook's data practices and whether the company violated a 2011 consent decree with the Federal Trade Commission that required the company to obtain explicit consent from users before sharing their personal information with third parties.

What might prompt a bipartisan coalition to sue Facebook? It isn't clear that there's any one thing. But lawmakers and regulators have been increasingly alarmed by the company's recent controversies, which include a data breach affecting more than 50 million users, the spread of fake news on the platform, and the spreading of propaganda by Russian-backed accounts during the 2016 election.

It's also not clear how serious the Democratic attorneys general are about filing a lawsuit. While the attorneys general announced their investigation in a press release, they said they had not yet reached a decision on whether to sue. In the past, some of the attorneys general have focused their efforts on trying to extract settlements from the tech companies without filing lawsuits.

"As attorneys general who have investigated and prosecuted health-care fraud, price-fixing, and other violations of our states' antitrust laws, we are committed to holding companies accountable when they fail to protect consumers and competition in the market," Ms. James said in a statement.

The investigation, which was first reported by The New York Times, was announced as the companies that dominate the tech industry are facing scrutiny from antitrust regulators around the world. Google, Twitter, Facebook and Amazon have faced a series of antitrust probes and class-action lawsuits in the United States and Europe ...

## C. Text Samples

Below are some text samples generated by Megatron-LM using a context prompt. Some of the texts are cut short.

**Context:** A flower, sometimes known as a bloom or blossom, is the reproductive structure found in flowering plants (plants of the division Magnoliophyta, also called angiosperms). The biological function of a flower is to effect reproduction, usually by providing a mechanism for the union of sperm with eggs.

**Megatron-LM:** The structural components of the flower are the petals and sepals, which are the reproductive organs, the hypanthium, which forms the base of the flower, and the calyx, which is the stigma. A flower is a composite structure, made up of sepals and petals, the sepals being composed of three fused tepals and the petals of two fused petals, though there are a few exceptions to this rule. The three sepals are fused to form a tube called the calyx, which is the structure that attracts the pollinating insects. The petals are composed of two fused sepals (the outer ones) which open and fold inward (clasp) at the tips, forming the petals. The four sepals and petals are united to form a corolla, the outermost structure of a flower that carries pollen.

The structure of the flower has important functional significance, including protection of the ovary from insects and the transfer of pollen between the male and female parts of a flower. The structure of the flower is also used by the plants to attract pollinators, for example by the shape of the petals and the colours of the stamens.

The structure of a flower varies widely between plants. The simplest flowers are unbranched and made up of only two sepals and a single petal. Other examples are composed of several petals joined together, with the innermost sepals completely fused to form a tube called the calyx. The petals are usually coloured to attract insects, which can only see certain wavelengths of light. The sepals are often coloured too, but usually have a more limited range of colours.

Some flowers have additional structures, such as stamens, anthers and ovaries. The presence of stamens and anthers in some flowers is used for cross-pollination. The ovary is the organ where the female gametes are produced, and is the first structure that is developed in an embryo of a flower. The ovary is the female flower's reproductive system, and consists of the ovary wall, the ovule, and the oviduct.

Flowers vary in their size and shape, from tiny to huge, and are often brightly coloured. Flowers may be seen as part of a larger structure, such as a flower head, an inflorescence, a bract, a spathe, or a carpels.

The term "flower" is usually applied to the floral structures that are visible externally, but many ...

**Context:** OpenAI did not release the full GPT-2 model due to concerns of malicious use, but they did release a smaller version equivalent in size to the original GPT (117 M parameters), trained on the new, larger dataset. Although not as powerful as the large model, the smaller version still has some language generation chops.

**Megatron-LM:** In December 2016, DeepMind released an even larger model, GPT-3, with more than 1.5 billion parameters. DeepMind has no plans to release the model, so the question remains whether other AI researchers will be able to copy and build on it.

Other language models have been built using other techniques. In 2016, a team of researchers at Columbia University, the University of Toronto, and Google DeepMind showed that neural network-based models, including deep neural networks, can learn the grammatical rules of a language. A group at Google DeepMind demonstrated that these rules could be represented in a network as a kind of decision tree, where the nodes in the tree are the words in the sentence, and the rules for the words are represented by the decision nodes at the end of the tree. The network would go through the network in order to choose the next word in the sentence based on whether it follows the rules or not. In 2017, the team at DeepMind released a paper detailing the training and performance of their model.

The problems with the model are many. One of the biggest issues is that grammatical rules are ambiguous. For example, the phrase "I came down" could mean I came down from the stairs, or I came down from the sky. In English, there are numerous other examples of sentences that are ambiguous. The model does not have any way to tell the difference between the ambiguous sentences, and so it simply makes a prediction about the entire sentence based on what rules it does know. This makes the model vulnerable to the phenomenon of overfitting. In order to avoid this, models must be trained using a large set of data, so that the model can find useful patterns in the data, and use those patterns to help make predictions. This approach is also prone to errors, as it relies on the training data to help make correct predictions, and the data might be incorrectly labeled, resulting in a model that produces a poor set of predictions. In the case of GPT-2, the model only has about 100 training sentences for each type of phrase (subject-verb-object).

Additionally, languages have different rules for different types of phrasing. In English, the subject-verb-object construction is fairly universal, but other languages have different rules, which makes training a GPT more complicated.

Finally, there is an inherent trade-off between the number of rules and the performance of the model ...

**Context:** The 36th International Conference on Machine Learning (ICML 2019) will be held in Long Beach, CA, USA from June 10th to June 15th, 2019. The conference will consist of one day of tutorials (June 10), followed by three days of main conference sessions (June 11-13), followed by two days of workshops (June 14-15).

**Megatron-LM:** With a broad scope, the conference addresses the challenges and opportunities in machine learning for practitioners and researchers. The conference program will feature sessions that cover different machine learning topics, from reinforcement learning to computational biology and from AI to deep learning. It is the world’s premier forum for exploring and presenting advances in machine learning.

ICML 2019 will offer an impressive range of sessions, which will be of interest to practitioners, researchers, and students. Some of the topics we will cover include:

Reinforcement learning

Soft Computing

Computational biology

Convolutional networks

Deep learning

Statistical learning theory

Statistical learning machine

Natural language processing

Biology

Awards

Since its first edition, the conference has grown and evolved. Today, it is the premier forum for showcasing state-of-the-art research in machine learning. The conference is widely recognized as the preeminent venue for showcasing the latest research in machine learning, and a place where practitioners, researchers, and students can exchange ideas and insights.

The conference is a peer-reviewed venue that draws its prestige and influence from the excellence of the presentations and the quality of the invited speakers. With over 50% of all talks selected based on their content by an expert committee, it is clear that the conference’s high standards of research and quality keep a very high bar.

**Context:** Rescue workers on Tuesday found a flight recorder from the Russian plane that crashed into the Black Sea over the weekend, the defense ministry said. All 92 people aboard the Russian military’s Tu-154 plane are believed to have died Sunday morning when it crashed two minutes after taking off from the southern Russian city of Sochi.

**Megatron-LM:** A child holds up a portrait of a dead relative at a makeshift morgue in a building belonging to a local government in the town of Petropavlovsk-Kamchatsky. A Russian Defense Ministry Tu-154 plane with 92 people on board crashed into the Black Sea on Sunday morning shortly after taking off from the southern Russian city of Sochi. (Yuri Kochetkov/EPA)

According to the ministry, the plane was found at a depth of more than 1,000 feet and the recorder was damaged. “The defense ministry has decided to hand over the recorder to the Interstate Aviation Committee, which is the main investigative body in the air crash,” the statement said.

The search for the plane’s black boxes is being hampered by strong winds and rough seas. Investigators have launched a criminal probe into the crash, which officials said was likely caused by a technical fault.

Russian President Vladimir Putin declared Wednesday a day of mourning for the victims.

## D. Further Scaling Analysis

In this section we study the effect of number of attention heads on the scaling results. We also present strong scaling results for our 1.2 billion parameter model.

### D.1. Attention Heads and Scaling

This section studies the effect of attention heads on model parallel scaling. To this end, we consider the 8.3 billion parameter configuration with 8-way model parallelism and vary the number of heads from 16 to 32. The results are presented in Table 7. As the number of attention heads increases, some of the GEMMS inside the self-attention layer become smaller and also the number of elements in the self attention softmax increases. This results in a slight decrease in scaling efficiency. Future research should be wary of this hyperparameter to design large transformer models that balance model speed and model accuracy.

### D.2. Strong Scaling

Our model parallelism is primarily designed to enable training models larger than what can fit in the memory of a

Table 7. Effect of number of attention heads on scaling on 8.3 billion of parameters with 8-way model parallelism.

Attention heads	Hidden size per head	Scaling Efficiency
16	192	82%
24	128	80%
32	96	77%

Table 8. Speedup obtained for the 1.2 billion parameters model using model parallelism while keeping the batch size constant.

# of GPUs	1	2	4	8
Speedup	1.0	1.64	2.34	2.98

single GPU, but it can also accelerate the training of smaller models without increasing the batch size. To measure this acceleration we train a model with a fixed 1.2 billion parameters. We use a fixed batch size of 8 samples per iteration and increase the number of GPUs using model parallelism. The results are listed in Table 8. Using two GPUs makes training 64% faster. Above that we see diminishing returns as the per-GPU computation decreases and the memory bandwidth and communication overheads begin to dominate.

## E. Evaluating Language Models Using WikiText103 and LAMBADA

In this section we detail our evaluation methodology for the WikiText103 dataset (Merity et al., 2016) and cloze-style prediction accuracy on the LAMBADA dataset(Paperno et al., 2016).

### E.1. Wikitext103 Perplexity

WikiText103 perplexity is an evaluation criterion that has been well studied over the past few years since the creation of the benchmark dataset. Perplexity is the exponentiation of the average cross entropy of a corpus (Mikolov et al., 2011). This makes it a natural evaluation metric for language models which represent a probability distribution over entire sentences or texts.

$$PPL = \exp\left(-\frac{1}{T_o} \sum_t^T \log P(t|0:t-1)\right) \quad (4)$$

To calculate perplexity in (4) we tokenize the WikiText103 test corpus according to our subword vocabulary and sum the cross entropy loss from each token  $[0, T]$ . We then normalize the cross entropy loss by the number of tokens in the original tokenization scheme  $T_o$ . The WikiText103 test corpus already comes pre-tokenized with word level tokens that prior works have used to compute perplexity. To evaluate our models’ perplexities on a level playing field with prior

works we must normalize by the original number of tokens,  $T_o$ , rather than the number of tokens,  $T$ , actually in the tokenized data fed as input to our model. This pre-tokenization also introduces artifacts in the text that are not present in our training data. To alleviate this distributional mismatch, we first preprocess the WikiText103 test dataset with invertible detokenizers to remove various artifacts related to punctuation and whitespace. The value of  $T_o$  is calculated before this preprocessing. For WikiText103’s test set  $T_o = 245566$  and  $T = 270329$ .

We must also make one further transformer-specific modification to the perplexity calculation. Unlike RNN-based language models, transformers operate on a fixed window input size. Therefore they cannot fully calculate  $P(t|0:t-1)$  and can only calculate  $P(t|t-w:t-1)$  where  $w$  is the size of our context: 1024 tokens. However, calculating this value for every token in our dataset is prohibitively expensive since we must compute approximately  $T$  evaluations of a  $w$  sized context. To evaluate our models efficiently we take a middle ground approach termed *overlapping evaluation* where we advance the sliding window by some overlap  $o$  each time and only compute the cross entropy losses corresponding to the last  $o$  tokens of the window. In our experiments we utilize an overlap  $o$  of 32, and compute losses over all sliding windows in such a fashion.

### E.2. LAMBADA Cloze Accuracy

The capability to handle long term contexts is crucial for state of the art language models and is a necessary prerequisite for problems like long-form generation and document-based question answering. Cloze-style datasets like LAMBADA are designed to measure a model’s ability to operate in and reason about these types of long term contexts. Cloze-style reading comprehension uses a context of word tokens  $x = x_{1:t}$  with one token  $x_j$  masked; the models objective is to correctly predict the value of the missing  $j^{\text{th}}$  token. To accurately predict the missing token, the model requires an in-depth understanding of the surrounding context and how language should be used in such a context. LAMBADA uses cloze-style reading comprehension to test generative left-to-right language models by constructing examples of 4-5 sentences where the last word in the context  $x_t$  is masked. Our models utilize subword units, so for LAMBADA evaluation we utilize the raw, unprocessed LAMBADA dataset and require that our model predict the multiple subword tokens that make up the word token. We use teacher forcing, and consider an answer correct only when all output predictions are correct. This formulation is equivalent to the original task of word token prediction.

# NVIDIA Tensor Core Programmability, Performance & Precision

Stefano Markidis, Steven Wei Der Chien, Erwin Laure

*KTH Royal Institute of Technology*

Ivy Bo Peng, Jeffrey S. Vetter

*Oak Ridge National Laboratory*

## Abstract

The NVIDIA Volta GPU microarchitecture introduces a specialized unit, called *Tensor Core* that performs one matrix-multiply-and-accumulate on  $4 \times 4$  matrices per clock cycle. The NVIDIA Tesla V100 accelerator, featuring the Volta microarchitecture, provides 640 Tensor Cores with a theoretical peak performance of 125 Tflops/s in mixed precision. In this paper, we investigate current approaches to program NVIDIA Tensor Cores, their performances and the precision loss due to computation in mixed precision.

Currently, NVIDIA provides three different ways of programming matrix-multiply-and-accumulate on Tensor Cores: the CUDA Warp Matrix Multiply Accumulate (WMMA) API, CUTLASS, a templated library based on WMMA, and cuBLAS GEMM. After experimenting with different approaches, we found that NVIDIA Tensor Cores can deliver up to 83 Tflops/s in mixed precision on a Tesla V100 GPU, seven and three times the performance in single and half precision respectively. A WMMA implementation of batched GEMM reaches a performance of 4 Tflops/s. While precision loss due to matrix multiplication with half precision input might be critical in many HPC applications, it can be considerably reduced at the cost of increased computation. Our results indicate that HPC applications using matrix multiplications can strongly benefit from using of NVIDIA Tensor Cores.

## Keywords

NVIDIA Tensor Cores; GPU Programming; Mixed Precision; GEMM

## I. INTRODUCTION

The raising markets of AI-based data analytics and deep-learning applications, such as software for self-driving cars, have pushed several companies to develop specialized hardware to boost the performance of large dense matrix (tensor) computation. This is essential to both training and inferencing of deep learning applications [1]. For instance, Google designed the Tensor Processing Unit [2] specifically for tensor calculations. Recently, NVIDIA released the Volta microarchitecture featuring specialized computing units called *Tensor Cores*.

An NVIDIA Tensor Core is capable of performing one matrix-multiply-and-accumulate operation on a  $4 \times 4$  matrix in one GPU clock cycle. In mixed-precision mode, Tensor Cores take input data in half floating-point precision, perform matrix multiplication in half precision and the accumulation in single precision.

The NVIDIA Tesla V100 GPU provides a total of 640 Tensor Cores that can reach a theoretical peak performance of 125 Tflops/s. Hence, systems like the NVIDIA DGX-1 system that combines eight Tesla V100 GPUs could achieve a theoretical peak performance of one Pflops/s in mixed precision. The pre-exascale systems, such as the Summit supercomputer that has six Tesla V100 GPUs connected with high-speed NVLink in each compute node for a total of 4,600 nodes, will offer nearly 18M Tensor Cores!

While large deep neural network applications will likely benefit from the use of NVIDIA Tensor Cores, it is still unclear how traditional HPC applications can exploit Tensor Cores. We envision three challenges. The first is to determine suitable programming models for Tensor Cores. In this work, we try to understand which programming interface can provide rich expressiveness while enabling maximum performance. The second is to quantify performance improvement from using Tensor Cores for various problem sizes and workloads. The third is to quantify the loss of precision when using mixed precision operations and to design techniques to improve the accuracy. We foresee these challenges will be of paramount importance for the HPC community as incoming supercomputers, such as Sierra and Summit, will be equipped with NVIDIA Tesla V100 GPUs. HPC applications running on these systems will need to take advantage of the NVIDIA Tensor Cores to reach maximum performance.

The objective of this paper is to evaluate the three challenges by providing an up-to-date study of NVIDIA Tensor Core. In particular, we focus on programmability, performance and precision loss in the context of HPC applications. We summarize our main contributions as follows:

- We survey current programming interfaces that perform tensor operations on NVIDIA Tensor Cores.
- We characterize the performance of NVIDIA Tensor Cores when computing large matrix multiplication and batched matrix multiplications. We compare them with the performance of the same operations on CUDA cores to quantify the performance boost.

- We quantify precision loss in matrix multiplication due to half precision matrix input with varying matrix sizes.
- We propose a technique to decrease precision loss in matrix multiplications on Tensor Cores at the cost of increased computation.

The paper is organized as follows. We first introduce previous works related to tensor architectures in Section II. We briefly describe NVIDIA Volta microarchitecture in Section III. We then present current approaches for programming NVIDIA Tensor Cores in Section IV. We focus on precision loss due to NVIDIA Tensor Core mixed-precision and possible methods to decrease precision loss in Section V. We describe our experimental set-up in Section VI and present the performance results in Section VII. Finally, Section VIII discusses the main results of this work and concludes the paper.

## II. RELATED WORK

AI-based data analytics and deep neural network applications have become increasingly important in recent years. These applications lead to rapid development of software and hardware that efficiently express and support tensor operations, which are fundamental for deep neural network applications. TensorFlow is among the most popular open-source programming framework that uses a computational graph with tensor operations as nodes of the graph [3]. Caffe, Torch and Microsoft CNTK are other popular programming frameworks for developing deep neural networks [1].

The seminal paper by Krizhevsky *et al.* [4] has established GPUs as the main workforce in training deep neural networks and triggered a renaissance of deep-learning applications. Besides NVIDIA Tensor Cores [5] discussed in this paper, several companies are also employing and developing specialized hardware for high-performance inference. Microsoft deployed the Catapult system that uses FPGAs [6]. Movidius developed the Myriad 2 Vision Processing Unit [7]. Google designed and developed Tensor Processing Unit (TPU) specifically for inference workloads. The main engine of the TPU is a MAC matrix multiply unit containing  $256 \times 256$  MACs, each capable of performing 8-bit multiply-and-adds on signed or unsigned integers. In December 2017, Intel announced the release of the Neural Network Processor (NPP) [8], which implements a new memory architecture for tensor operations. NPP does not have standard caches and data movement is programmable by software. In addition, neuromorphic hardware, such as the IBM TrueNorth [9] and SpiNNaker [10] chips, mimics the functioning of spiking neural network. Although their original design purpose is to simulate the brain, they may also find usage in AI applications.

These new architectures usually have lower power and energy footprint than general processors that are employed in traditional HPC systems. The reduced power consumption mainly comes from reduced accuracy in computation by using fewer bits for representation. In deep neural networks, floating-points are transformed to narrow precision via quantization. For instance, TPU operates on eight-bit integers. NVIDIA Tensor Cores follows the IEEE 754 standard [11] and uses mixed floating-point precision, i.e., matrix multiplication input in half precision and accumulation in single precision. Intel NPP introduces a new format, called *Flexpoint* [12]. Flexpoint uses fixed-point multiplications and a shared exponent to allow a large dynamic range. While several studies have shown that deep neural networks are tolerant to low precision calculation [13], [14], [15], such studies are still in their infancies in HPC. Mixed single and double precision calculations have been studied in the context of HPC [16], [17]. However, these emerging architectures have rather narrow precision, smaller than single precision, and the topic is still to be studied in details.

## III. NVIDIA VOLTA ARCHITECTURE

In May 2017, NVIDIA released Volta GV100 GPU architecture and the Tesla V100 accelerator to boost AI and HPC applications. As one of the largest silicon chips, Volta GPU includes 21.1 billion transistors on a die area of  $815 \text{ mm}^2$ .

A full GV100 GPU consists of six GPU Processing Clusters (GPCs). Each GPC contains seven Texture Processing Clusters (TPCs) and 14 Streaming Multiprocessors (SMs). A 16 GB HBM2 memory, connecting through eight memory controllers in four memory stacks, is embedded in the same package. We present the architecture of GV100 GPU in the simplified diagram of Fig. 1.

The Volta microarchitecture features a renewed Streaming Multiprocessor (SM) design [18] (Figure 2). Each SM is partitioned into four processing blocks. Each block consists of two Tensor Cores, 8 FP64 cores, 16 FP32 cores, 16 INT32 cores and one Special Function Unit (SFU). One main design change in Volta SM is the integration of L1 data cache and shared memory subsystem. Their combined capacity of 128 KB per SM is  $7 \times$  larger than the data cache of Volta's predecessor GP100 GPU. Also, texture units and SMs can share this merged L1 cache/shared memory and configure up to 96 KB shared memory per SM. The Tesla V100 accelerator uses 80 SMs for a total of 2,560 FP64 cores, 5,120 FP32 cores and 640 Tensor Cores.

A new feature of Volta SM is mixed-precision operations with Tensor Cores. In each cycle, a Tensor Core can perform 64 floating-point Fused-Multiply-Add (FMA) operations [5]. An FMA operation takes input values in half precision while the output values can be either in half (FP16) or full precision (FP32) as illustrated in Fig. 3. FMA has the advantage of using only one rounding operation instead of two, resulting in a more accurate output [11].

In total, the Tesla V100 accelerator can perform up to 40,960 FMA operations per cycle, i.e., 81,920 floating-point operations. The Tesla V100 accelerator uses the base clock frequency 1.3 GHz and it can be boosted to 1.53 GHz. The theoretical maximum performance can reach 31.4 Tflops/s with half precision, 15.7 Tflops/s with single precision, and 7.8 Tflops/s with double precision while Tensor Cores can deliver 125 Tflops/s.

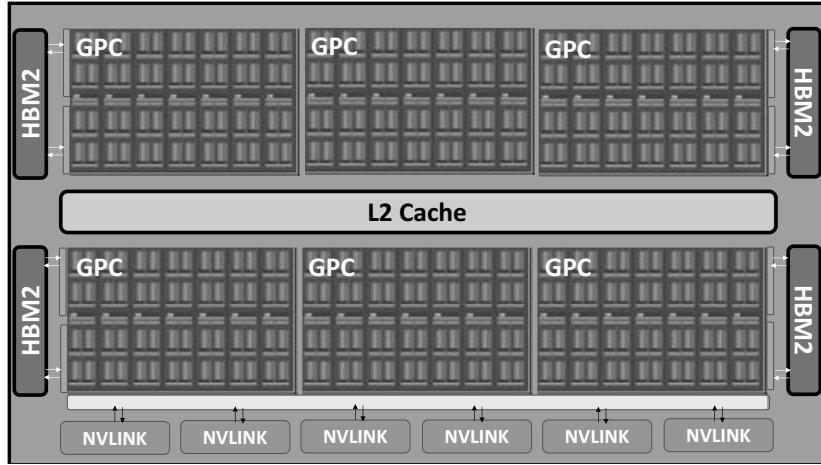


Fig. 1: Volta GV100 GPU architecture features six GPCs and 16 GB HBM2. Adapted from [18].

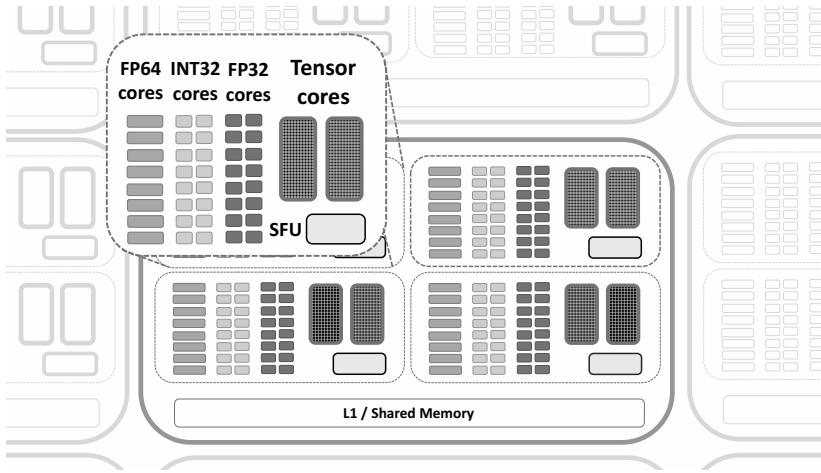


Fig. 2: Simplified diagram of the Volta SM architecture. The NVIDIA Tesla V100 uses 80 SMs.

Half precision data and instruction in Tensor Cores are the main contributors to high throughput. Compared to single precision, half precision data only requires half memory bandwidth and footprint, resulting in faster data transfer.

#### IV. PROGRAMMING NVIDIA TENSOR CORES

The NVIDIA Tensor Core basically performs only one kind of operation: matrix-multiply-and-accumulate on  $4 \times 4$  matrices. Therefore, a programming interface for NVIDIA Tensor Cores can simply express the BLAS GEMM (GEneral Matrix to Matrix Multiplication) operation. A GEMM operation consists of the multiplication of two matrices  $A$  and  $B$  and accumulation of the result into a third matrix  $C$ , i.e.  $C = \alpha AB + \beta C$ . Here we present different interfaces of Tensor Cores to illustrate their programmability with different levels of abstraction.

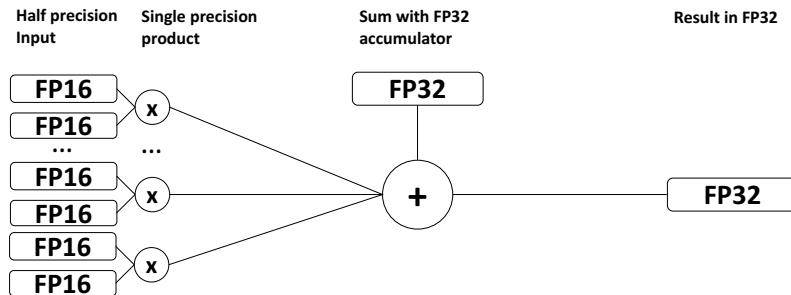


Fig. 3: FMAs in NVIDIA Tensor Cores.

Currently, the lowest level interface to program NVIDIA Tensor Cores is CUDA 9 Warp Matrix Multiply and Accumulation (WMMA) API. CUDA 9 WMMA is a CUDA preview feature and WMMA will likely be changed in future releases with no backward compatibility guarantee. We briefly present it as at the moment it is the only way to program Tensor Cores directly and future APIs might be developed upon CUDA 9 WMMA.

CUDA 9 allows us to program a basic matrix-multiply-and-accumulate on  $16 \times 16$  matrices. Recent CUDA 9 releases, such as CUDA 9.1, also support non-square matrix multiplication with different sizes. We note that while NVIDIA Tensor Core implements  $4 \times 4$  matrix multiplications in hardware, CUDA 9 WMMA allows us only to compute larger matrix multiplications. This is in-line with the CUDA philosophy of running many more threads than hardware computing units (problem over-decomposition) to hide instruction and memory latencies.

Listing 1: CUDA 9 WMMA provides a direct way to calculate  $16 \times 16$  matrix matrix-multiply-and-accumulate using a CUDA Warp (32 threads).

```
// Calculate AB with NVIDIA Tensor Cores
// Kernel executed by 1 Warp (32 Threads)
__global__ void tensorOp(float *D, half *A, half *B) {
    // 1. Declare the fragments
    wmma::fragment<wmma::matrix_a, M, N, K, half, wmma::col_major> Amat;
    wmma::fragment<wmma::matrix_b, M, N, K, half, wmma::col_major> Bmat;
    wmma::fragment<wmma::accumulator, M, N, K, float, void> Cmat;
    // 2. Initialize the output to zero
    wmma::fill_fragment(Cmat, 0.0f);
    // 3. Load the inputs into the fragments
    wmma::load_matrix_sync(Amat, A, M);
    wmma::load_matrix_sync(Bmat, B, K);
    // 4. Perform the matrix multiplication
    wmma::mma_sync(Cmat, Amat, Bmat, Cmat);
    // 5. Store the result from fragment to global
    wmma::store_matrix_sync(D, Cmat, M, wmma::mem_col_major);
}
```

Listing 1 presents a CUDA kernel that performs a matrix multiplication of two  $16 \times 16$  matrices with one CUDA Warp (32 threads). The kernel consists of five parts. First, the WMMA fragments (GPU register memory for storing the input matrices) `Amat`, `Bmat` and `Cmat` are declared. Second, the accumulator fragment, `Cmat`, for storing the result of the matrix multiply, is set to zero. Third, the input matrices are loaded into the fragments `Amat`, `Bmat` using `wmma::load_matrix_sync()`. Fourth, the multiplication is performed by calling the `wmma::mma_sync()`. Finally, we move the results from the fragment `Cmat` to `D` in the GPU global memory. Each matrix multiplication and accumulation should be executed by one CUDA Warp (32 threads). If the kernel `tensorOp` is launched with less than 32 threads, the result of the matrix multiplication is undetermined. On the other hand, using more threads than a Warp will still result in the correct results.

An important point is that the two-dimensional tensors are provided as 1-D arrays. For this reason, we need to declare if the 1-D arrays should be interpreted either as row- or column-major.

#### A. Matrix Multiplication

While CUDA 9 WMMA provides a direct way of performing GEMM only with fixed-size matrices, three other methods can be used to calculate matrix multiplications of arbitrary size:

- **Tiled Matrix Multiply with CUDA 9 WMMA.** With this technique, the result matrix,  $C$  is divided in fixed-size tiles (sub-matrices), i.e.  $16 \times 16$ , and each of the  $C$  tile values can be calculated by summing the result of  $A$  and  $B$  tile multiplications. This tiling technique for matrix multiplication is widely used in GPU programming to exploit shared GPU memory. One thread block per tile is used [19] while in the case of CUDA 9 WMMA, a Warp is assigned to the tile.
- **NVIDIA CUTLASS** (CUDA Templates for Linear Algebra Subroutines) is a CUDA C++ templated header-only library to perform GEMM operation in different precisions (`dgemm`, `sgemm` and `hgemm`) [20]. It supports also CUDA 9 WMMA implementation (`wgemm`). The library supports different tiling strategies and exploits software pipelining to hide GPU memory latencies.
- **NVIDIA cuBLAS** is an NVIDIA library that implements standard basic linear algebra subroutines (BLAS) [21]. The library provides GEMM routines for Tensor Cores. In order to perform GEMM on NVIDIA tensor Cores, the cuBLAS math mode needs to be set to `CUBLAS_tensorOp_MATH` using the function `cublasSetMathMode()`. It is then possible to use either `cublasGemmEx()` or `cublasSgemm()` to perform GEMM on NVIDIA tensor Cores.

### B. Batched Matrix Multiplications

Many HPC applications rely on the solution of several small-size matrix multiplications in parallel [22]. One example is the Nek5000 CFD application that uses small-size matrix multiplies for each spectral element resulting from the semi-spectral discretization [23], [24]. In this case, the matrix size depends on the order of the spectral element in each direction. Another application is the Fast Multipole Method-accelerated Fast Fourier Transform (FFT) that requires also many small matrix multiplications [25]. BLAS GEMM routines are optimized for solving large matrix multiplications and do not perform optimally in solving small-size matrix multiplications. Libraries, such as LIBXSMM [26] and Intel MKL, provide high-performance small-size matrix multiplications.

The most convenient approach to solve several small matrix multiplication in parallel on GPU is through NVIDIA cuBLAS. The NVIDIA cuBLAS library provides a batched `sgemm` API for single precision matrix multiply, called `cublasSgemmBatched()`. However, batched `GEMM` is not supported by NVIDIA Tensor Cores<sup>1</sup>. In this work, we implement a simple batched `GEMM`, based on Listing 1, to evaluate the possible performance benefit of using NVIDIA Tensor cores to solve batched `GEMM`.

### V. PRECISION LOSS

Each Tensor Core performs a multiplication of two matrices with half precision floating-point entries and adds the result to an accumulator in single precision (see Fig. 3). The use of mixed precision calculations might cause large rounding errors, affecting simulation accuracy.

One of the motivations for matrix multiplication in half precision is that the matrix entries that are multiplied in neural network are small with respect to the value of the previous iteration. For this reason, the multiplication result is still small in value. However, the result is accumulated to another value that might be much larger. To avoid precision loss or use additional computation, i.e. Kahan summation [28], accumulation is performed in single precision.

In addition, deep neural network training are tolerant to precision loss up to certain degree [13], [14], [15]. Thus, high precision calculations are not critical for the completion of many deep neural network trainings. On the other hand, the vast majority of traditional HPC applications, with probably the exception of Montecarlo codes, are considerably more sensitive to rounding errors that arise from the usage of narrow precision. For this reason, it is important to characterize the impact of mixed precision calculations in widely used HPC computational kernels, such as `GEMM`. Narrow precision matrix multiplications might severely impact the possible usage of NVIDIA Tensor Cores in HPC applications. Half precision floating-point representation uses 16 bits: one bit for the sign, five bit for the exponent and ten bits for the significand (or fraction or mantissa), as illustrated in Fig. 4.

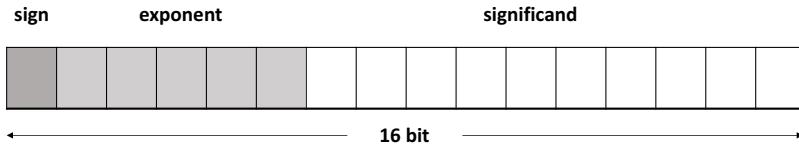


Fig. 4: Half precision floating-point number representation.

The limited number of bits of floating-point number representation introduces two limitations:

- **Limited range.** Having only five bits of exponent, the maximum representable number in half precision is 65,504 and the range of half precision floating-point is  $\pm 65,504$ . When rounding a value from single to half precision, if the float number is larger than 65,504, it is set to half infinity. The machine epsilon in half precision floating numbers is  $2^{-10}$ . Any float number that is too small to be represented as a half will be set to zero.
- **Decreasing precision with increasing value range intervals.** The most striking characteristic of using half precision is the extreme precision loss for large numbers. In fact, when using half precision, we have only 1,024 values for each power of two number intervals. For instance, there are 1,024 representable numbers between one ( $2^0$ ) and two ( $2^1$ ). Between 1,024 ( $2^{10}$ ) and 2048 ( $2^{11}$ ), there are also only 1,024 values so all the fractional precision is lost for numbers larger than 1,024. For the same reason, there is only an accuracy of  $\pm 32$  between 32,768 ( $2^{15}$ ) and 65,536 ( $2^{16}$ ).

It is clear from this brief discussion that precision strongly depends on the value range of the numbers used in simulations: rounding relatively large number from single to half precision leads to considerable precision loss.

While the impact of half precision input on large HPC applications requires in-depth studies [29], we observe that it is possible to decrease the precision loss in matrix multiplications,  $C = AB$ , at the cost of increased computation and memory consumption with a simple technique.

<sup>1</sup>After the completion of this work, batched `GEMM` API for Tensor Cores was released in cuBLAS 9.1.128, among other optimizations [27].

We define a half-precision residual matrix  $R$  as the difference between a matrix before and after rounding from single to half-precision, where  $A_{single}$  and  $A_{half}$  represents a matrix before and after rounding (notation is analogous with matrix  $B$ ):

$$R_A = A_{single} - A_{half}. \quad (1)$$

We manipulate  $A_{single}B_{half}$  and compute on Tensor Cores using the distributive property of matrix multiplication and sum as

$$A_{single}B_{half} = (A_{single} - A_{half} + A_{half})B_{half} = (R_A + A_{half})B_{half} = R_AB_{half} + A_{half}B_{half}. \quad (2)$$

The equation above allows us to take into account the rounding error from single to half-precision for matrix  $A$  with one additional matrix multiplication on NVIDIA Tensor Cores and additional memory for storing  $R_A$ . We call this simple technique *precision refinement*, as it is similar to analogous techniques, called *iterative precision refinement* and used in other works for the solution of linear systems [29].

Since  $B_{half}$  is still rounded directly from  $B_{single}$ , precision loss is only partially eliminated. It is possible to further recover precision by applying the same technique again to matrix  $B$  with  $R_B = B_{single} - B_{half}$  and apply the distributive property of matrix multiplication and sum:

$$A_{single}B_{single} = (R_A + A_{half})(R_B + B_{half}) = R_AR_B + A_{half}R_B + R_AB_{half} + A_{half}B_{half}. \quad (3)$$

In this case, we can reduce the precision loss by performing four matrix multiplications on the NVIDIA Tensor Cores and using additional memory for storing  $R_A$  and  $R_B$ .

We motivate our method by the assumption that the precision loss due to conversion arises from the fact that 16-bit cannot entirely represent all values in 32-bit. Thus, we distribute the un-representable portion of the value (residual) to another 16-bit number. Since the value is originally in 32-bit, it can be fully represented by two 16-bit numbers, subject to error from distribution. In other words, we recover the loss in precision due to input conversion by additional operation with residual values that were recorded during conversion. With this scheme, depending on the precision requirement of an application, the developer can choose to perform refinement on one or both matrices at the expense of additional computation time and memory.

## VI. EXPERIMENTAL SET-UP

We test NVIDIA Tensor Cores with a Tesla V100 accelerator which is connected to an Intel E5-2690v3 Haswell host. The Operating System is CentOS Linux version 7.4.1708. We use CUDA Driver/Runtime Version 9.0 with CUDA Capability 7.0. The GNU compiler version for compiling host code is 4.8.5. The nvcc compiler flags `-O3 -Xptxas -v -std=c++11 -gencode arch=compute_70,code=sm_70 -gencode arch=compute_70,code=compute_70` are used. The tested Tesla V100 supports a base default GPU clock at 1.245 GHz and a boost GPU clock at 1.38 GHz. In this paper, we report the results using the boost GPU clock at 1.38 GHz. We note that the GPU boost frequency in our system is 10% lower than the GPU boost frequency reported in Ref. [18]. With GPU clock at 1.38 GHz, the theoretical peak performance on Tensor Cores is 112.7 Tflops/s.

We measure performance of Tensor Cores using GEMM operation,  $C = \alpha AB + \beta C$  with  $\alpha = 1.0$  and  $\beta = 1.0$ . We initialize  $A$ ,  $B$  and  $C$  values in single floating-point precision. When the GEMM is computed on the Tensor Cores, the values of  $A$  and  $B$  are first rounded to half precision. The time to complete the rounding is not considered when timing the overall execution time.

We report the results using square matrices with size  $N$  for each dimension. We take Tflops/s as the main figure of merit for performance. To time the CUDA execution of kernels running on the GPU, we use CUDA events that have a resolution of approximately half microsecond. The number of operations are calculated assuming that the matrix multiplication uses the naive algorithm requiring  $\mathcal{O}(N^3)$  operations. We note that cuBLAS GEMM matrix multiplication might use other matrix multiplication algorithms, i.e. Strassen algorithm. In this case, the performance of the cuBLAS might be affected by the algorithm in use also.

We run 5 to 100 tests and present the harmonic mean of flops/s in the plots. If the execution time is taken as the performance figure of merit, we report the arithmetic mean of execution times. We do not show error bars when the error is less than 1%.

For the sake of comparison, we also report the performance of a naive implementation using CUDA 9 WMMA without any optimization (see Listing 1), such as the use of CUDA shared memory and software pipeline. CUTLASS makes use of these techniques to provide an optimized use of CUDA 9 WMMA. When we measure the performance of CUTLASS, we tested different tiling techniques with different execution configurations; we report the timing of the set-up with higher performance for a given  $N$ .

As NVIDIA does not provide yet a batched GEMM for Tensor Cores <sup>1</sup>, we wrote a simple implementation for testing purposes, extending the code in Listing 1. For batched GEMM, we only use square  $16 \times 16$  matrices. In this case, the CUDA execution configuration consists of 512 threads per block. Since a  $16 \times 16$  matrix multiplication is executed by one Warp (32 threads), 16 matrix multiplications are executed per thread block.

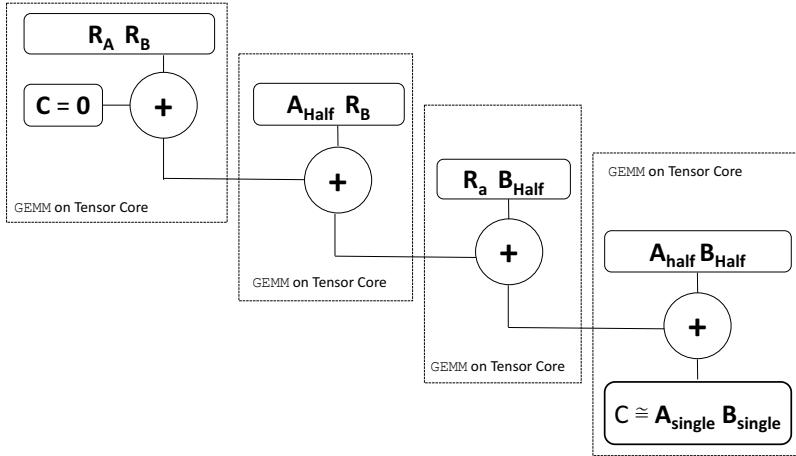


Fig. 5: Implementation of precision refinement using four pipelined GEMM on Tensor Cores.

To quantify the precision loss due to mixed precision computations, we first calculate the error matrix  $e$  as  $e = (C_{\text{half}} - C_{\text{single}})$ , where  $C_{\text{half}}$  is the result of the matrix multiplication with half precision input and  $C_{\text{single}}$  is the result with single precision input. We then apply the max norm  $\|e\|_{\text{Max}} = \max(|e_{i,j}|)$ . We choose the max norm to quantify the error as it provides a bound of the maximum error per matrix entry. We initialize the two square matrices  $A$  and  $B$  of size  $N$  with random numbers, taken from range [-1,1] in single precision. The matrix values are then converted to half precision. We then vary the matrix size  $N$  to study how the total number of operations affects the overall precision loss.

In addition, we implement Eq. 2 (precision refinement with  $R_A$ ) and Eq. 3 (precision refinement with both  $R_A$  and  $R_B$ ) to assess the computational cost of techniques to reduce the precision loss. The diagram in Fig. 5 shows the implementation of Eq. 3 using four pipelined GEMM to perform matrix multiplications on Tensor Cores. In this case, we use a quick implementation based on four cuBLAS function calls such that the result of a GEMM is used as half precision input for the next GEMM. We note that optimized versions of such techniques are possible. We provide a simple implementation for fast comparison and estimation of the computational cost for decreasing the precision loss.

## VII. RESULTS

In this section, we present and discuss the experimental results. Our results show that using NVIDIA Tensor Cores to compute GEMM can lead to considerable performance boost. Fig. 6 presents the GEMM performance with and without Tensor Cores. The bars in white color show the GEMM performance with CUDA cores in full single and half precision without Tensor Cores. The bars in grey color show the GEMM performance on Tensor Cores using a naive implementation with CUDA 9 WMMA, CUTLASS and cuBLAS respectively. In addition, a line at 112.7 Tflops/s (theoretical peak using Tensor Cores in our system) is superimposed to the plot.

### A. Performance

We achieved maximum performance of 83 Tflops/s on NVIDIA Tensor Cores for  $N = 8,192$  using cuBLAS GEMM. The measured peak performance in mixed precision is approximately 74% the theoretical performance of the NVIDIA Tensor Cores, which is about  $6\times$  and  $3\times$  the performance of GEMM in full single and half precision. For  $N = 16,384$ , CUTLASS performs better than cuBLAS GEMM on Tensor Cores. This is probably due to the fact that CUTLASS can be tested with different tiling configurations to select the most performant setup.

The naive CUDA 9 WMMA implementation does not provide any performance improvement with respect to sgemm on the CUDA cores. Also, it is outperformed by the hgemm in half precision. If the GEMM implementation with CUDA 9 WMMA also includes the use of CUDA shared memory, the performance (not shown here) is about five times higher than the performance of the naive implementation for  $N = 8,192$ . This indicates that it is critical to use CUDA shared memory to reduce memory traffic [19] when programming NVIDIA Tensor Cores.

We also evaluate the potential performance improvement when running batched GEMM on Tensor Cores. We compare the performance of the cuBLAS batched sgemm in single precision on CUDA cores with the performance of a simple implementation of batched GEMM using CUDA 9 WMMA on Tensor Cores. Fig. 7 shows a box plot of the batched GEMM performance with and without Tensor Cores in white and grey boxes. The number of  $16\times 16$  matrix multiplies, or *batch size*, is represented on the  $x$  axis of the plot, while the performance in Tflops/s is shown on the  $y$  axis. The measured peak performance is 4 Tflops/s for 262,144 matrix multiplications with half precision input on Tensor Cores. Increasing the number of  $16\times 16$  matrix multiplies increases the performance of the GEMM with and without Tensor Cores. When using cuBLAS batched sgemm for *batchsize* > 131,072,

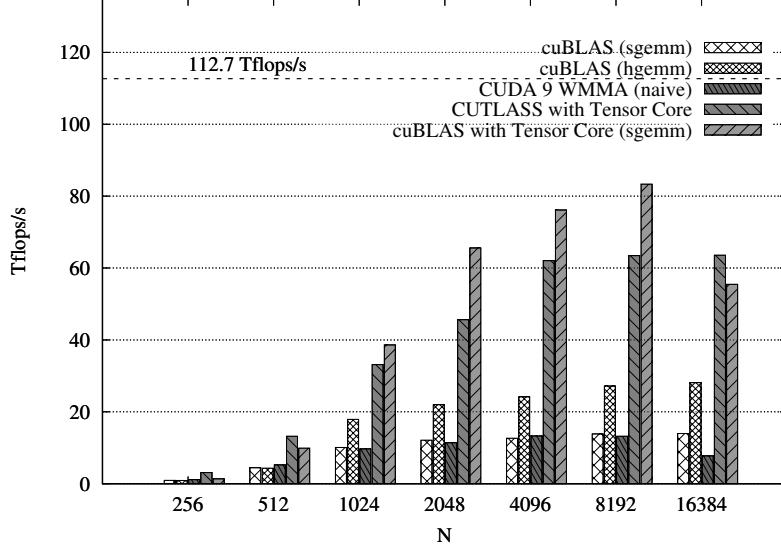


Fig. 6: GEMM performance without Tensor Cores in single and half precision (white bars) and with Tensor Cores using naive implementation with CUDA 9 WMMA, CUTLASS and cuBLAS (grey bars) varying with matrix size  $N$ .

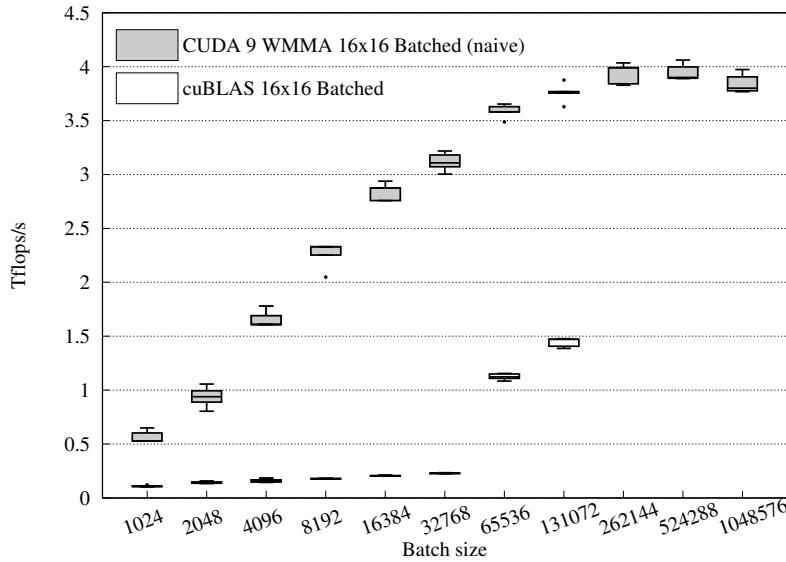


Fig. 7: Performance of cuBLAS batched sgemm on CUDA cores, and CUDA 9 WMMA implementation performing batch size  $16 \times 16$  matrix multiplies. The cuBLAS batched sgemm cannot run for more than 131,072 multiplications as they require more memory than the available one on the Tesla V100 GPU.

the system runs out of memory. For this reason, results for cuBLAS batched sgemm for  $batchsize > 131,072$  are not reported in the plot. The performance of our naive implementation of batched GEMM with half precision inputs outperforms the cuBLAS batched sgemm in full single precision. The performance of batched GEMM varies between  $2.5\times$  and  $12\times$  the performance of cuBLAS batched sgemm varying the batch size.

### B. Precision and refinement

We first measure the precision loss by half precision input on Tensor Cores. We then use Eqs. 2 and 3 to quantify the decrease of precision loss. Fig. 8 shows the error  $\|e\|_{Max}$  for multiplications on Tensor Cores (white bars) varying with matrix size. Using the techniques in Eq. 2 (light gray bars) and Eq. 3 (dark gray bars), the precision loss can be decreased. It is clear from Fig. 8 that by increasing the matrix size  $N$ , rounding error increases. This is due to the fact that the number of multiplications and summations for calculating one matrix element scales as  $N^2$ . So, the error scales quadratically with  $N$ .

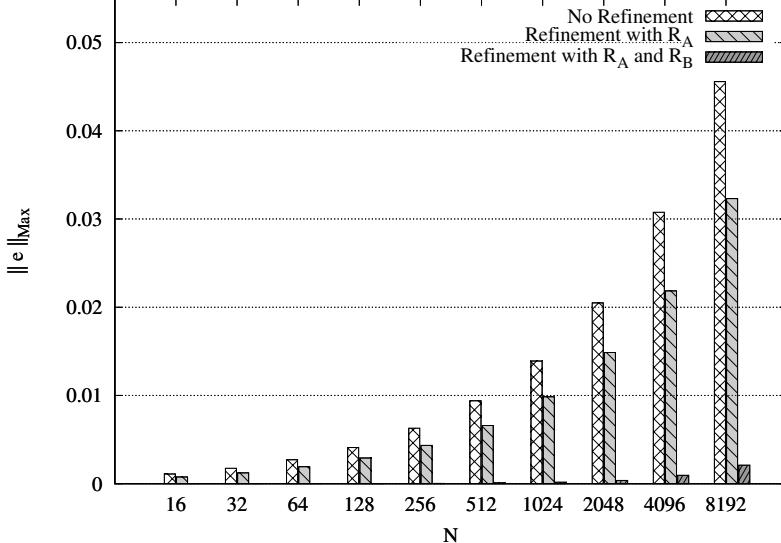


Fig. 8: Error in half precision (no refinement, white bars), using precision refinement with  $R_A$ , and precision refinement with both  $R_A$  and  $R_B$  varying the matrix size  $N$ .

From Fig. 8 we can see that the use of Eq. 2 is only partially beneficial: we observe a 30% decrease of the error for  $N = 8,192$ . This small error decrease is due the fact that the norm of the two matrices is approximately the same. The use of Eq. 3 is more effective in decreasing the precision: the error is decreased by a factor of ten for  $N = 8,192$ . We note that the precision loss strongly depends on matrix input values. For instance, if the  $A$  and  $B$  values are chosen randomly between  $\pm 16$  and  $N = 4,096$ , we measure  $\|e\|_{Max} = 8.32$  for  $AB$  with no refinement, and  $\|e\|_{Max} = 0.24$  for  $AB$  with  $A$  and  $B$  refinement (Eq. 3). In this case, the use of the refinement leads to a  $35\times$  decrease of error.

Finally, we quantify the computational cost of applying the refinement technique discussed in Section V to decrease the precision loss when using NVIDIA Tensor Cores. Fig. 9 presents a scatter plot in the execution time vs error plane for  $8,192 \times 8,192$  and  $4,096 \times 4,096$  matrix multiplication on Tensor Cores (square symbols), using precision refinement with  $R_A$  (circle symbols) and with both  $R_A$  and  $R_B$  (triangle symbols). The scatter plot points are spread in the *error* direction because we use random input values uniformly distributed between minus one and one as matrix entries. On the other hand, execution time measurements show little variation. In addition to scatter plot points, we add two lines at 10 and 80 ms to represent the average execution time recorded to perform a matrix multiplication in full single precision for  $N = 4,096$  and  $N = 8,192$ , for which the error  $\|e\|_{Max}$  is zero.

It is clear from Fig. 9 that by increasing the computational cost (execution time) the error decreases. For  $N = 8,192$ , if we increase the computational cost of a factor of 2.25, we can obtain a reduction of precision loss of approximately 30% using precision refinement only with  $R_A$ . The precision refinement with both  $R_A$  and  $R_B$  leads to an error that is approximately  $10\times$  smaller than initial error with a  $5\times$  computational cost. The computational cost of precision refinement with  $R_A$  and  $R_B$  is still approximately 25% lower than the cost of completing a GEMM without Tensor Cores. We note the implementation of precision refinement using four pipelined GEMM on Tensor Cores shown in 5 is not optimized as the precision refinement takes more than four times the time of completing one GEMM. For this reason, there is room for a large performance improvement.

## VIII. DISCUSSION AND CONCLUSIONS

The NVIDIA V100 GPUs will be an important asset for the upcoming supercomputers, providing a large fraction of their overall computing power. The Volta microarchitecture features for the first time Tensor Cores, specially designed to perform tensor operations. We showed that the use of NVIDIA Tensor cores can boost the GEMM performance by  $6\times$  when multiplying large matrices and  $2.5\times$ - $12\times$  when multiplying many small-size matrices in parallel.

Many HPC applications are based on the multiplication of large matrices or several small-size matrix in parallel. For this reason, such applications can take direct advantage of NVIDIA Tensor Cores. On the other hand, some HPC applications, such particle-based codes, might require a reformulation of the algorithms in tensorial form to use the Tensor Cores effectively. In addition, a more in-depth study of the impact of mixed precision calculation on the overall simulation accuracy in large HPC applications is required to promote the uptake of NVIDIA Tensor cores by HPC applications.

In this paper, we focused on three main aspects when using NVIDIA Tensor Cores in HPC applications: programmability, performance and precision loss. We summarize our findings for each aspect as follows.

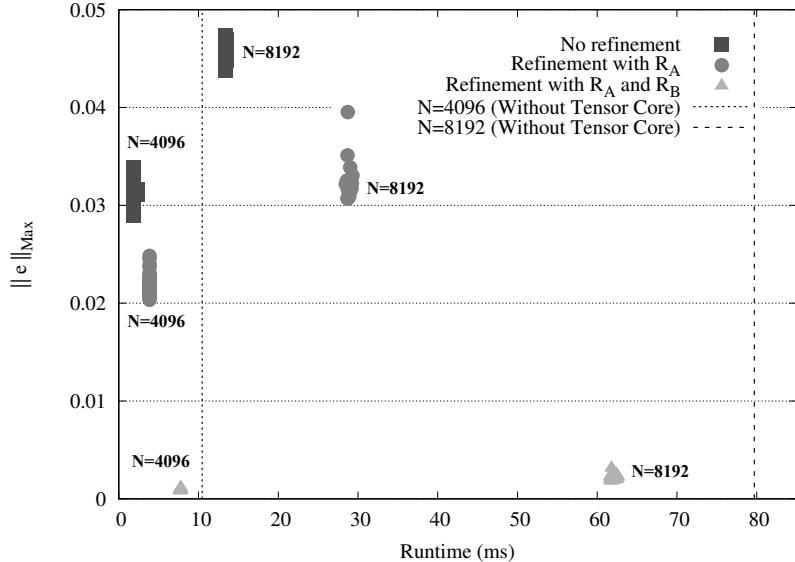


Fig. 9: Scatter plot with  $\|e\|_{Max}$  on  $x$  axis and runtime on the  $y$  axis for GEMM with no refinement (squares), refinement with only  $R_A$  (circles), and with both  $R_A$  and  $R_B$  (triangles) for  $N = 8,192$  and  $N = 4,096$ . The two dashed lines represent the execution time for sgemm without Tensor Cores.

- **Programmability.** Currently, there are three programming interfaces for developing applications using matrix-multiply-and-accumulate on NVIDIA Tensor Cores. CUDA 9 WMMA API provides direct access to CUDA Tensor Cores and can be used in combination. However, WMMA is a preview feature and will likely be modified in future releases. The other two ways of programming NVIDIA Tensor Cores are via CUTLASS and cuBLAS libraries. The CUTLASS implementation is based on WMMA and provides different tiling sizes that can be used for performance tuning. The NVIDIA cuBLAS library allows the use of Tensor Cores by setting cuBLAS math mode to CUBLAS\_tensorOp\_MATH. In this work, we have not covered support for convolution operations on Tensor Cores by the NVIDIA cuDNN [30], a library of primitives for deep neural networks because we focus on HPC usage of Tensor Cores. However, many of the concepts we covered in this paper can also be applied to cuDNN.

Finally, we note that from a high-level point of view, NVIDIA Tensor Cores can be seen as accelerators within an accelerator. Tensor Cores not only add considerable computing performance boost (a factor seven for GEMM), they also work on low precision and have their own local memory consisting of *fragments*. When designing future interfaces for NVIDIA Tensor Cores, one possibility would be to treat Tensor Cores as accelerator-in-accelerator and investigate the use of direct kernel launching on NVIDIA tensor cores from the host.

- **Performance.** We achieved the maximum performance in our test environment at 83 Tflops/s with cuBLAS GEMM. The naive GEMM implementation with CUDA WMMA did not lead to any performance improvement; however when using the implementation with CUDA shared memory, we measured a  $5\times$  performance improvement with respect to the sgemm performance on CUDA cores (not shown here). The problem size for maximum performance with GEMM is  $N = 8,192$ . We also implemented a batched GEMM for Tensor Cores using CUDA 9 WMMA to evaluate the potential benefit of solving several small matrix multiplications in parallel. Although the implementation is not optimized, NVIDIA Tensor Cores still provided a performance increase of  $2.5\times - 12\times$  with respect to performance of the cuBLAS batched sgemm. This shows that NVIDIA Tensor Cores could also be used to perform small-size matrix multiplications efficiently.

When investigating possible performance optimization, we noted that memory traffic still has high impact on the overall performance of the matrix multiplications, despite Volta's integration of L1 data cache and shared memory subsystem. For this reason, application optimization for NVIDIA tensor cores is likely to include strategies for data placement on the GPU memory subsystem.

An additional optimization is to use CUDA cores and Tensor Cores concurrently. This can be achieved by using CUDA streams in combination with CUDA 9 WMMA. This will also allow for more advanced and optimized pipelined mixed precision refinement methods and implementations.

- **Precision.** As matrix multiplication inputs are in half precision, precision loss occurs when they are rounded from single to half precision. In particular, precision loss is considerable when using large input values. When input matrix size increases, error increases because  $\mathcal{O}(N^2)$  operations are required to calculate a matrix entry in the matrix multiplication. We showed a simple method to decrease precision loss by taking into account the rounding error when converting values

from single to half precision. This method reduces the precision loss at the cost of increased computation. Further methods for increasing the precision can be developed, possibly taking advantage of single precision computation of unused CUDA cores while performing tensor operations.

In conclusion, despite the Volta microarchitecture with Tensor Cores has only been recently released, it is possible to program Tensor Cores for HPC applications using three approaches and achieve considerable performance boost at the cost of decreased precision. We expect the programming interfaces for NVIDIA Tensor Cores to evolve and allow increased expressiveness and performance. In particular, we noted that the measured Tensor Core maximum performance is still 74% the theoretical peak, leaving room for improvement. While the precision loss due to mixed precision might be an obstacle for the uptake of NVIDIA Tensor Cores in HPC, we showed that it is possible to decrease it at the cost of increased computations. For all these reasons, it is very likely that HPC applications will strongly benefit from using of NVIDIA Tensor Cores. In the future we will focus on testing Tensor Cores in real-world HPC applications, such as Nek5000 [23], [24] or Fast Multipole Method-accelerated FFT [25].

#### ACKNOWLEDGEMENT

This work used computing resources from KTH PDC Center for High Performance Computing. The Authors would like to thank Daniel Ahlin and Gilbert Netzer for their assistance when using the Tesla V100 at PDC. Funding for the work is received from the European Commission H2020 program, Grant Agreement No. 671500 (SAGE).

#### REFERENCES

- [1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.
- [2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 1–12.
- [3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “TensorFlow: A system for large-scale machine learning,” in *OSDI*, vol. 16, 2016, pp. 265–283.
- [4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
- [5] NVIDIA, “NVIDIA Tesla V100 GPU architecture,” 2017, accessed: 2018-01-27. [Online]. Available: <http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>
- [6] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, “A reconfigurable fabric for accelerating large-scale datacenter services,” in *Computer Architecture (ISCA)*. IEEE, 2014, pp. 13–24.
- [7] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O’Riordan, and V. Toma, “Always-on vision processing unit for mobile applications,” *IEEE Micro*, vol. 35, no. 2, pp. 56–66, 2015.
- [8] K. Carey, “Intel Nervana™ Neural Network Processor: Architecture Update,” 2017, accessed: 2017-01-27. [Online]. Available: <https://ai.intel.com/intel-nervana-neural-network-processor-architecture-update/>
- [9] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura *et al.*, “A million spiking-neuron integrated circuit with a scalable communication network and interface,” *Science*, vol. 345, no. 6197, pp. 668–673, 2014.
- [10] M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras, and S. B. Furber, “SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor,” in *Neural Networks, 2008.*, 2008, pp. 2849–2856.
- [11] N. Whitehead and A. Fit-Florea, “Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs,” 2011, accessed: 2017-01-27. [Online]. Available: <https://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf>
- [12] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Hall, L. Horof, A. Khosrowshahi *et al.*, “Flexpoint: An adaptive numerical format for efficient training of deep neural networks,” in *Advances in Neural Information Processing Systems*, 2017, pp. 1740–1750.
- [13] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaev, G. Venkatesh *et al.*, “Mixed precision training,” *arXiv preprint arXiv:1710.03740*, 2017.
- [14] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” in *International Conference on Machine Learning*, 2015, pp. 1737–1746.
- [15] M. Courbariaux, J.-P. David, and Y. Bengio, “Low precision storage for deep learning,” *arXiv preprint arXiv:1412.7024*, 2014.
- [16] A. Buttari, J. Dongarra, J. Langou, P. Luszczek, and J. Kurzak, “Mixed precision iterative refinement techniques for the solution of dense linear systems,” *The International Journal of High Performance Computing Applications*, vol. 21, no. 4, pp. 457–466, 2007.
- [17] A. Haidar, P. Wu, S. Tomov, and J. Dongarra, “Investigating half precision arithmetic to accelerate dense linear system solvers,” in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ACM, 2017, p. 10.
- [18] L. Durant, O. Giroux, M. Harris, and N. Stam, “Inside Volta: The world’s most advanced data center GPU,” 2017, accessed: 2017-01-27. [Online]. Available: <https://devblogs.nvidia.com/inside-volta/>
- [19] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.
- [20] A. Kerr, D. Merrill, J. Demouth, and J. Tran, “CUTLASS: Fast linear algebra in CUDA C++,” 2017, accessed: 2017-01-27. [Online]. Available: <https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/>
- [21] NVIDIA, “cuBLAS library,” *NVIDIA Corporation, Santa Clara, California*, vol. 15, no. 27, p. 31, 2008.
- [22] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon, “The design and performance of batched BLAS on modern high-performance computing systems,” *Procedia Computer Science*, vol. 108, pp. 495 – 504, 2017, international Conference on Computational Science, ICCS 2017.
- [23] N. Offermans, O. Marin, M. Schanen, J. Gong, P. Fischer, P. Schlatter, A. Obabko, A. Peplinski, M. Hutchinson, and E. Merzari, “On the strong scaling of the spectral element solver Nek5000 on petascale systems,” in *Proceedings of the Exascale Applications and Software Conference 2016*. ACM, 2016, p. 5.
- [24] S. Markidis, J. Gong, M. Schliephake, E. Laure, A. Hart, D. Henty, K. Heisey, and P. Fischer, “OpenACC acceleration of the Nek5000 spectral element code,” *The International Journal of High Performance Computing Applications*, vol. 29, no. 3, pp. 311–319, 2015.
- [25] C. Cecka, “Low communication FMM-accelerated FFT on GPUs,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 54.
- [26] A. Heincke, G. Henry, M. Hutchinson, and H. Pabst, “LIBXSMM: accelerating small matrix multiplications by runtime code generation,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, p. 84.

- [27] “NVIDIA CUDA toolkit release notes,” accessed: 2018-03-01. [Online]. Available: <http://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html>
- [28] N. J. Higham, “The accuracy of floating point summation,” *SIAM Journal on Scientific Computing*, vol. 14, no. 4, pp. 783–799, 1993.
- [29] P. Luszczek, J. Kurzak, I. Yamazaki, and J. Dongarra, “Towards numerical benchmark for half-precision floating point arithmetic,” in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–5.
- [30] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “cuDNN: Efficient primitives for deep learning,” *arXiv preprint arXiv:1410.0759*, 2014.

---

# Online normalizer calculation for softmax

---

**Maxim Milakov**

NVIDIA

[mmilakov@nvidia.com](mailto:mmilakov@nvidia.com)

**Natalia Gimelshein**

NVIDIA

[ngimelshein@nvidia.com](mailto:ngimelshein@nvidia.com)

## Abstract

The Softmax function is ubiquitous in machine learning, multiple previous works suggested faster alternatives for it. In this paper we propose a way to compute classical Softmax with fewer memory accesses and hypothesize that this reduction in memory accesses should improve Softmax performance on actual hardware. The benchmarks confirm this hypothesis: Softmax accelerates by up to 1.3x and Softmax+TopK combined and fused by up to 5x.

## 1 Introduction

Neural networks models are widely used for language modeling, for tasks such as machine translation [1] and speech recognition [2]. These models compute word probabilities taking into account the already generated part of the sequence. The probabilities are usually computed by a Projection layer, which "projects" hidden representation into the output vocabulary space, and a following Softmax function, which transforms raw logits into the vector of probabilities. Softmax is utilized not only for neural networks, for example, it is employed in multinomial logistic regression [3].

A number of previous works suggested faster alternatives to compute word probabilities. Differentiated Softmax [4] and SVD-Softmax [5] replace the projection layer - which is usually just a matrix multiplication - with more computationally efficient alternatives. Multiple variants of Hierarchical Softmax [6, 7, 8] split a single Projection+Softmax pair into multiple much smaller versions of these two functions organized in tree-like structures. Sampled-based approximations, such as Importance Sampling [9], Noise Contrastive Estimation [10], and Blackout [11] accelerate training by running Softmax on select elements of the original vector. Finally, Self-Normalized Softmax [12] augments the objective function to make the softmax normalization term close to 1 (and skip computing it during inference).

This is not an exhaustive list, but, hopefully, a representative one. Almost all of the approaches still need to run the original Softmax function, either on full vector or reduced one. There are two exceptions that don't need to compute the softmax normalization term: training with Noise Contrastive Estimation and inference with Self-Normalized Softmax. All others will benefit from the original Softmax running faster.

To the best of our knowledge there has been no targeted efforts to improve the performance of the original Softmax function. We tried to address this shortcoming and figured out a way to compute Softmax with fewer memory accesses. We benchmarked it to see if those reductions in memory accesses translate into performance improvements on a real hardware.

## 2 Original softmax

Function  $y = \text{Softmax}(x)$  is defined as:

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^V e^{x_j}} \quad (1)$$

where  $x, y \in \mathbb{R}^V$ . The naive implementation (see algorithm 1) scans the input vector two times - one to calculate the normalization term  $d_V$  and another to compute output values  $y_i$  - effectively doing three memory accesses per vector element: two loads and one store.

---

### Algorithm 1 Naive softmax

---

```

1:  $d_0 \leftarrow 0$ 
2: for  $j \leftarrow 1, V$  do
3:    $d_j \leftarrow d_{j-1} + e^{x_j}$ 
4: end for
5: for  $i \leftarrow 1, V$  do
6:    $y_i \leftarrow \frac{e^{x_i}}{d_V}$ 
7: end for
```

---

Unfortunately, on real hardware, where the range of numbers represented is limited, the line 3 of the algorithm 1 can overflow or underflow due to the exponent. There is a safe form of (1), which is immune to this problem:

$$y_i = \frac{e^{x_i - \max_{k=1}^V x_k}}{\sum_{j=1}^V e^{x_j - \max_{k=1}^V x_k}} \quad (2)$$

All major DL frameworks are using this safe version for the Softmax computation: TensorFlow

---

### Algorithm 2 Safe softmax

---

```

1:  $m_0 \leftarrow -\infty$ 
2: for  $k \leftarrow 1, V$  do
3:    $m_k \leftarrow \max(m_{k-1}, x_k)$ 
4: end for
5:  $d_0 \leftarrow 0$ 
6: for  $j \leftarrow 1, V$  do
7:    $d_j \leftarrow d_{j-1} + e^{x_j - m_V}$ 
8: end for
9: for  $i \leftarrow 1, V$  do
10:   $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$ 
11: end for
```

---

[13] v1.7, PyTorch [14] (with Caffe2) v0.4.0, MXNET [15] v1.1.0, Microsoft Cognitive Toolkit [16] v2.5.1, and Chainer [17] v5.0.0a1. But Safe Softmax does three passes over input vector: The first one calculates the maximum value  $m_V$ , the second one - normalization term  $d_V$ , and the third one - final values  $y_i$ , see algorithm 2; This results in 4 memory access per vector element overall. We want to improve on that.

## 3 Online normalizer calculation

The algorithm 3 calculates both the maximum value  $m$  and the normalization term  $d$  in a single pass over input vector with negligible additional cost of two operations per vector element. It reduces memory accesses from 4 down to 3 per vector element for the Softmax function evaluation. Inspiration came from the numerically stable variance calculation online algorithm, see [18].

---

**Algorithm 3** Safe softmax with online normalizer calculation

---

```

1:  $m_0 \leftarrow -\infty$ 
2:  $d_0 \leftarrow 0$ 
3: for  $j \leftarrow 1, V$  do
4:    $m_j \leftarrow \max(m_{j-1}, x_j)$ 
5:    $d_j \leftarrow d_{j-1} \times e^{m_{j-1}-m_j} + e^{x_j-m_j}$ 
6: end for
7: for  $i \leftarrow 1, V$  do
8:    $y_i \leftarrow \frac{e^{x_i-m_V}}{d_V}$ 
9: end for

```

---

Essentially, the algorithm keeps the maximum value  $m$  and the normalization term  $d$  as it iterates over elements of the input array. At each iteration it needs to adjust the normalizer  $d$  to the new maximum  $m_j$  and only then add new value to the normalizer.

**Theorem 1.** *The lines 1-6 of the algorithm 3 compute  $m_V = \max_{k=1}^V x_k$  and  $d_V = \sum_{j=1}^V e^{x_j-m_V}$*

*Proof.* We will use a proof by induction.

◊ *Base case:*  $V = 1$

$$\begin{aligned}
m_1 &\leftarrow x_1 && \text{by line 4 of the algorithm 3} \\
&= \max_{k=1}^1 x_k \\
d_1 &\leftarrow e^{x_1-m_1} && \text{by line 5 of the algorithm 3} \\
&= \sum_{j=1}^1 e^{x_j-m_1}
\end{aligned}$$

The theorem holds for  $V = 1$ .

◊ *Inductive step:* We assume the theorem statement holds for  $V = S - 1$ , that is the lines 1-6 of the algorithm 3 compute  $m_{S-1} = \max_{k=1}^{S-1} x_k$  and  $d_{S-1} = \sum_{j=1}^{S-1} e^{x_j-m_{S-1}}$ . Let's see what the algorithm computes for  $V = S$

$$\begin{aligned}
m_S &\leftarrow \max(m_{S-1}, x_S) && \text{by line 4 of the algorithm 3} \\
&= \max(\max_{k=1}^{S-1} x_k, x_S) && \text{by the inductive hypothesis} \\
&= \max_{k=1}^S x_k \\
d_S &\leftarrow d_{S-1} \times e^{m_{S-1}-m_S} + e^{x_S-m_S} && \text{by line 5 of the algorithm 3} \\
&= \left( \sum_{j=1}^{S-1} e^{x_j-m_{S-1}} \right) \times e^{m_{S-1}-m_S} + e^{x_S-m_S} && \text{by the inductive hypothesis} \\
&= \sum_{j=1}^{S-1} e^{x_j-m_S} + e^{x_S-m_S} \\
&= \sum_{j=1}^S e^{x_j-m_S}
\end{aligned}$$

The inductive step holds as well. □

The algorithm 3 is proved to compute the Softmax function as defined in (2). It is also safe:

- $m_j$  is the running maximum,  $m_j \in \left[ \min_{k=1}^V m_k, \max_{k=1}^V m_k \right]$ ,  $\forall j \in 1, V$ ;  $m_j$  cannot underflow or overflow.

- $d_j$  is also bounded:  $1 \leq d_j \leq j, \forall j \in 1, V$ . It can be easily proven by induction. The 32-bit floating point storage for  $d_j$  guarantees processing of up to  $1.7 * 10^{37}$  elements in vector  $x$  without overflow. It is a reasonably large amount, but if your vector is even larger you need to use the 64-bit floating point storage for  $d_j$ .

The algorithm 2 provides the same guarantees:  $1 \leq d_j \leq j, \forall j \in 1, V$ .

In the remainder of this paper we will call algorithm 3 "Online Softmax".

### 3.1 Parallel online normalizer calculation

The lines 1-6 of the algorithm 3 define a sequential way of calculating the normalization term in a single pass over input vector. Modern computing devices allow running multiple threads concurrently; We need to have a parallel version of the algorithm to fully utilize devices. We define a generalized version of the online normalizer calculation:

$$\begin{bmatrix} m_V \\ d_V \end{bmatrix} = \begin{bmatrix} x_1 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} x_2 \\ 1 \end{bmatrix} \oplus \dots \oplus \begin{bmatrix} x_V \\ 1 \end{bmatrix} \quad (3)$$

where  $x_i, m_V, d_V \in \mathbb{R}$ . The binary operation  $\oplus : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^2$  is defined as:

$$\begin{bmatrix} m_i \\ d_i \end{bmatrix} \oplus \begin{bmatrix} m_j \\ d_j \end{bmatrix} = \begin{bmatrix} \max(m_i, m_j) \\ d_i \times e^{m_i - \max(m_i, m_j)} + d_j \times e^{m_j - \max(m_i, m_j)} \end{bmatrix} \quad (4)$$

Applying (3) sequentially from left to right is equivalent to running lines 1-6 of the algorithm 3. The operation  $\oplus$  is associative, which enables parallel evaluation of (3). It is also commutative, which provides the flexibility needed to make parallel implementations more efficient. We omit the proofs for these two statements for brevity.

## 4 Softmax and top-k fusion

Online Softmax (algorithm 3) does three memory accesses per vector element: one load for the normalizer calculation, one load and one store for computing Softmax function values  $y_i$ . Inference with the beam search for auto-regressive models has TopK following Softmax, and this TopK doesn't need to compute all  $y_i$  values. This enables even bigger improvements.

The TopK function is producing the vector of K integer indices referencing the largest values in the input vector, along with those values:

$$TopK(y) = (v, z) : v_i = y_{z_i}, v_i \geq y_j, \forall i \in [1, K], \forall j \notin z \quad (5)$$

where  $y \in \mathbb{R}^V, z \in \mathbb{Z}^K, v \in \mathbb{R}^K$ .

The TopK needs to load each element of the input vector at least once. Running Safe Softmax and the TopK separately requires 5 accesses per input element and 4 accesses if we use Online Softmax instead of Safe Softmax (but still run them separately, one after another). If we improve on the algorithm 3 and keep not only running values of  $m$  and  $d$  (when iterating over the input vector), but also the vectors of TopK input values  $u$  and their indices  $p$  - as in the algorithm 4 - we can run this Softmax+TopK fusion with just one memory access per element of the input vector.

## 5 Benchmarking

Online normalizer calculation reduces the number of memory accesses for the Softmax and Softmax+TopK functions. The softmax function has a very low flops per byte ratio; that means the memory bandwidth should be limiting the performance, even for Online Softmax with its additional few floating point operations per element. Fewer memory accesses should translate into performance improvements, and experiments confirm this.

We implemented a benchmark for GPUs using CUDA C. The benchmark utilizes CUB v1.8.0 for fast parallel reductions. All experiments were run on NVIDIA Tesla V100 PCIe 16 GB,

---

**Algorithm 4** Online softmax and top-k

---

```

1:  $m_0 \leftarrow -\infty$ 
2:  $d_0 \leftarrow 0$ 
3:  $u \leftarrow \{-\infty, -\infty, \dots, -\infty\}^T, u \in \mathbb{R}^{K+1}$     ▷ The 1st  $K$  elems will hold running TopK values
4:  $p \leftarrow \{-1, -1, \dots, -1\}^T, p \in \mathbb{Z}^{K+1}$                                 ▷ ... and their indices
5: for  $j \leftarrow 1, V$  do
6:    $m_j \leftarrow \max(m_{j-1}, x_j)$ 
7:    $d_j \leftarrow d_{j-1} \times e^{m_{j-1}-m_j} + e^{x_j-m_j}$ 
8:    $u_{K+1} \leftarrow x_j$                                 ▷ Initialize  $K+1$  elem with new value from input vector
9:    $p_{K+1} \leftarrow j$                                 ▷ ... and its index
10:   $k \leftarrow K$       ▷ Sort  $u$  in descending order, permuting  $p$  accordingly. The first  $K$  elements are
     already sorted, so we need just a single loop, inserting the last element in the correct position.
11:  while  $k \geq 1$  and  $u_k < u_{k+1}$  do
12:    swap( $u_k, u_{k+1}$ )
13:    swap( $p_k, p_{k+1}$ )
14:     $k \leftarrow k - 1$ 
15:  end while
16: end for
17: for  $i \leftarrow 1, K$  do                                ▷ The algorithm stores only  $K$  values and their indices
18:    $v_i \leftarrow \frac{e^{u_i-m_V}}{d_V}$ 
19:    $z_i \leftarrow p_i$ 
20: end for

```

---

ECC on, persistent mode on, CUDA Toolkit 9.1. Source code of the benchmark is available at [github.com/NVIDIA/online-softmax](https://github.com/NVIDIA/online-softmax).

### 5.1 Benchmarking softmax

We benchmarked all 3 Softmax algorithms - Naive, Safe, and Online - on different vector sizes for the batch sizes of 4,000 and 10. The large batch case corresponds to the training or batch inference with enough input vectors to saturate the device and the small batch case corresponds to online inference with too few vectors to occupy the device fully.

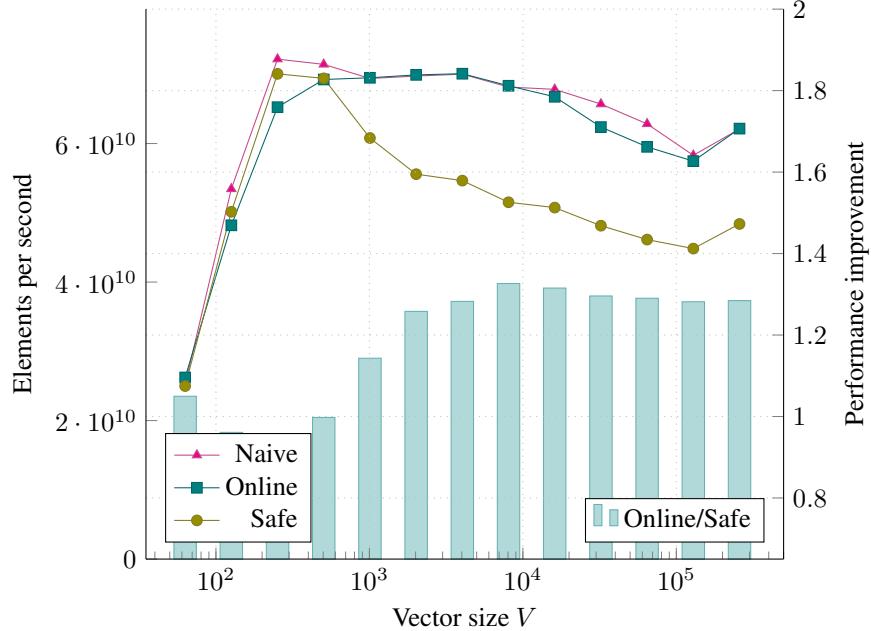


Figure 1: Benchmarking softmax, Tesla V100, fp32, batch size 4000 vectors

For the large batch case (see figure 1) all three algorithms perform similarly up until  $V = 1000$  vector size. The NVIDIA Visual Profiler shows that at that point L1 and L2 cache thrashing starts to make all three algorithms limited by the DRAM bandwidth. When this happens Online and Naive algorithms are getting faster than Safe one, quickly achieving  $\sim 1.3x$  at  $V = 4000$  (look for bars in the chart, they are showing performance improvement of Online Softmax over Safe Softmax). This is quite close to  $1.33x$  reduction in memory accesses for those algorithms.

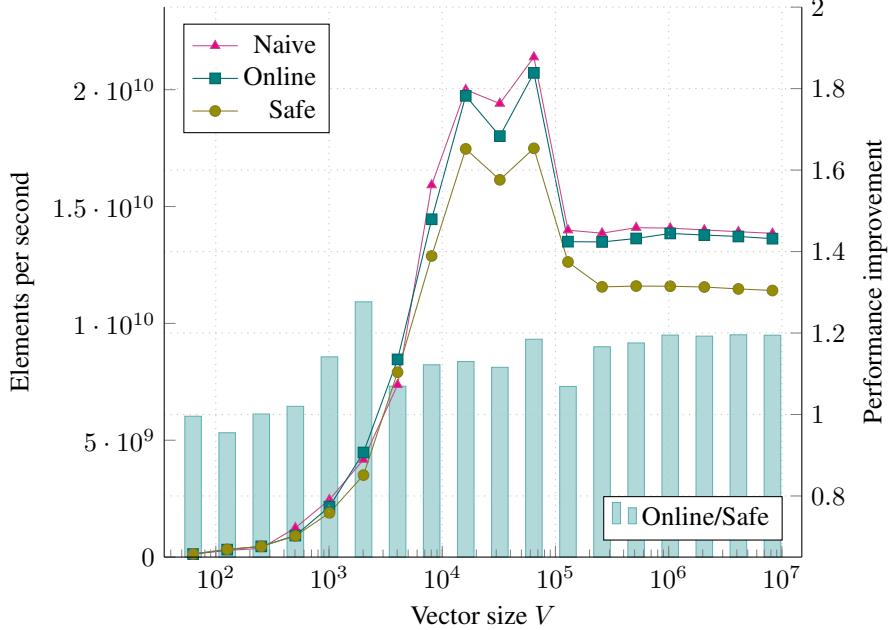


Figure 2: Benchmarking softmax, Tesla V100, fp32, batch size 10 vectors

The absolute performance for small batch case is lower for all algorithms, see figure 2. The benchmark is running one threadblock per vector; thus small batch case - with 10 vectors - has just 10 threadblocks in the grid. This is not enough to saturate the GPU, both compute and the memory subsystem are underutilized, various latencies are exposed. As in the batch inference case, all three algorithms show similar performance up to  $V = 1000$  vector size. After that Naive and Online algorithms outperform Safe one by  $\sim 1.15x$ .

## 5.2 Benchmarking softmax and top-k

We benchmarked Safe Softmax followed by the TopK (running one after another), Safe Softmax fused with the TopK into a single function, and Online Softmax fused with TopK, again, for 2 cases: 4,000 and 10 vectors. We picked up  $K = 5$  in TopK for all runs.

Online fused version is running considerably faster than Safe unfused one. For large batch case - see figure 3 - the performance improvement starts at  $1.5x$  and goes up as vector size  $V$  increases approaching  $5x$  at  $V = 25000$ , which corresponds to  $5x$  reduction in memory accesses. This  $5x$  comes from  $2.5x$  due to function fusion and  $2x$  due to Online Softmax itself.

In the small batch case (see figure 4) Online fused version outperforms Safe unfused one by  $1.5x$ - $2.5x$ . It cannot achieve  $5x$  because the GPU is underutilized and the performance is limited not by the memory bandwidth, but by various latencies. Yet the reduction in memory accesses helps even in this latency limited case. In small batch case fusion only already brings substantial performance improvements, switching to Online Softmax helps improve performance even further.

The benchmark shows these levels of performance improvement for relatively small  $K$  only. The cost of keeping partial TopK results - as in the lines 10-15 of the algorithm 4 - increases quickly as  $K$  gets bigger: the performance improvement drops to  $3.5x$  for  $K = 10$ ,  $2x$  for  $K = 15$ ,  $1.4x$  for  $K = 30$ , and degrades further for bigger  $K$ s. For these cases the TopK is dominating (in terms of runtime) over the Softmax and fusing the normalization term

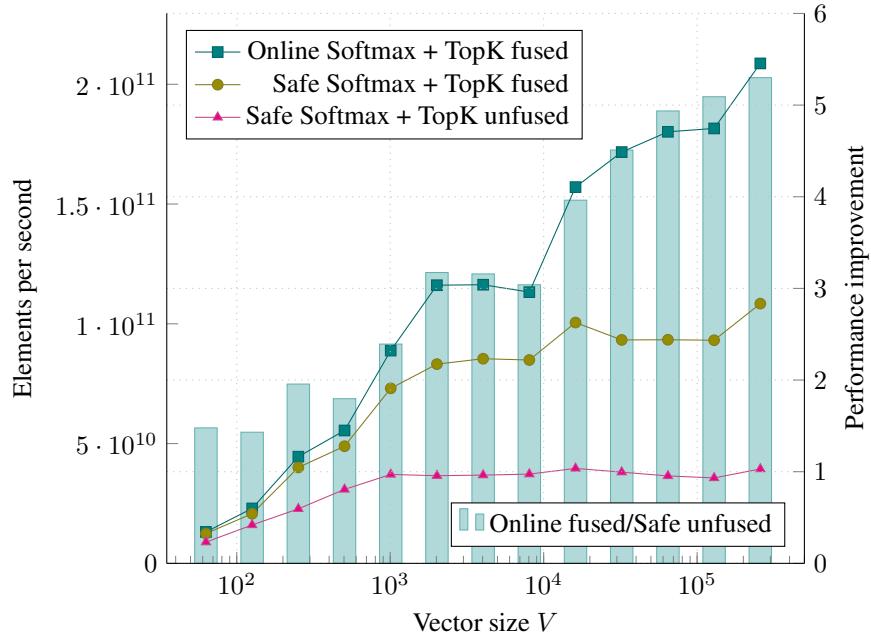


Figure 3: Benchmarking softmax and top-k, Tesla V100, fp32, batch size 4000 vectors

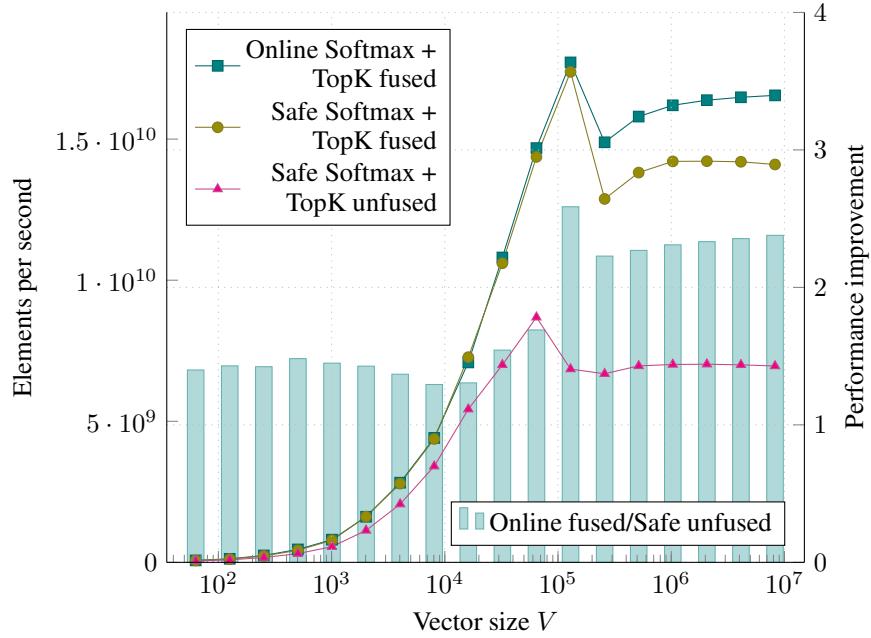


Figure 4: Benchmarking softmax and top-k, Tesla V100, fp32, batch size 10 vectors

calculation into the TopK is still beneficial, but the value goes down as TopK is taking more and more time.

## 6 Results

We introduced the way to calculate the normalizer for the Softmax function in a single pass over input data, which reduces memory accesses by 1.33x for the Softmax function alone. Benchmarks

on Tesla V100 show that this materializes in 1.15x performance improvements for  $V \geq 1000$  vector sizes, and for the large batch mode it goes up to 1.3x when  $V \geq 4000$ .

If one is using Naive Softmax then switching to Online version improves numerical accuracy with no performance hit or a negligible one.

When the TopK follows the Softmax the new single-pass normalizer calculation enables efficient fusion of these 2 functions resulting in 5x fewer memory accesses for Softmax+TopK combined. We observed 1.5x-5x performance improvement on Tesla V100, with this 5x improvement coming from 2.5x with fusion and 2x with Online Softmax itself.

These performance improvements could be applied not only to the classical Softmax function; They are orthogonal to many other Softmax optimization techniques including Hierarchical Softmax, Importance Sampling, and SVD-Softmax.

## 7 Discussion

Online Softmax is running up to 1.3x faster on the latest generation GPU than the one used by major DL frameworks. It also enables very efficient fusion of the Softmax with following TopK showing up to 5x performance improvement over the traditional Safe Softmax and TopK running separately.

Could we see significantly different speed-ups or even slow-downs on different compute devices, for example CPUs? We didn't do experiments for those, but if the original code is vectorized and one manages to keep it vectorized for the online normalizer (and partial TopK) calculation then similar speedups could probably be expected.

There could be a way to improve the performance further. The resulting Softmax and even Softmax+TopK fused are still limited by the memory bandwidth, so fusing them with the preceding layer will avoid memory round trip, thus improving performance. This change is more challenging though.

### Acknowledgments

We would like to thank Christoph Angerer for his valuable comments and suggestions.

## References

- [1] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural Networks. *ArXiv e-prints*, September 2014, 1409.3215.
- [2] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov 2012. ISSN 1053-5888.
- [3] Engel J. Polytomous logistic regression. *Statistica Neerlandica*, 42(4):233–252.
- [4] W. Chen, D. Grangier, and M. Auli. Strategies for Training Large Vocabulary Neural Language Models. *ArXiv e-prints*, December 2015, 1512.04906.
- [5] Kyuhong Shim, Minjae Lee, Iksoo Choi, Yoonho Boo, and Wonyong Sung. Svd-softmax: Fast softmax approximation on large vocabulary neural networks. In *Advances in Neural Information Processing Systems 30*, pages 5463–5473. Curran Associates, Inc., 2017. URL <http://papers.nips.cc/paper/7130-svd-softmax-fast-softmax-approximation-on-large-vocabulary-neural-networks.pdf>
- [6] Joshua Goodman. Classes for fast maximum entropy training. In *ICASSP*, pages 561–564. IEEE, 2001. ISBN 0-7803-7041-4. URL <http://dblp.uni-trier.de/db/conf/icassp/icassp2001.html#Goodman01>.
- [7] T. Mikolov, S. Kombrink, L. Burget, J. Černocký, and S. Khudanpur. Extensions of recurrent neural network language model. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5528–5531, May 2011.

- [8] E. Grave, A. Joulin, M. Cissé, D. Grangier, and H. Jégou. Efficient softmax approximation for GPUs. *ArXiv e-prints*, September 2016, 1609.04309.
- [9] Yoshua Bengio and Jean-Sébastien Sénécal. Quick training of probabilistic neural nets by importance sampling. In *Proceedings of the conference on Artificial Intelligence and Statistics (AISTATS)*, 2003.
- [10] A. Mnih and Y. Whye Teh. A Fast and Simple Algorithm for Training Neural Probabilistic Language Models. *ArXiv e-prints*, June 2012, 1206.6426.
- [11] S. Ji, S. V. N. Vishwanathan, N. Satish, M. J. Anderson, and P. Dubey. BlackOut: Speeding up Recurrent Neural Network Language Models With Very Large Vocabularies. *ArXiv e-prints*, November 2015, 1511.06909.
- [12] Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, and John Makhoul. Fast and robust neural network joint models for statistical machine translation. In *Proceedings of ACL2014*, pages 1370–1380, 2014.
- [13] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL <https://www.tensorflow.org/>. Software available from tensorflow.org.
- [14] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.
- [15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. URL <https://github.com/dmlc/web-data/raw/master/mxnet/paper/mxnet-learningsys.pdf>.
- [16] Frank Seide and Amit Agarwal. Cntk: Microsoft’s open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’16, pages 2135–2135, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4232-2. URL <http://doi.acm.org/10.1145/2939672.2945397>.
- [17] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. URL [http://learningsys.org/papers/LearningSys\\_2015\\_paper\\_33.pdf](http://learningsys.org/papers/LearningSys_2015_paper_33.pdf).
- [18] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962. URL <https://amstat.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022>.

# Efficient Memory Management for Large Language Model Serving with *PagedAttention*

Woosuk Kwon<sup>1,\*</sup> Zhuohan Li<sup>1,\*</sup> Siyuan Zhuang<sup>1</sup> Ying Sheng<sup>1,2</sup> Lianmin Zheng<sup>1</sup> Cody Hao Yu<sup>3</sup>  
Joseph E. Gonzalez<sup>1</sup> Hao Zhang<sup>4</sup> Ion Stoica<sup>1</sup>

<sup>1</sup>UC Berkeley <sup>2</sup>Stanford University <sup>3</sup>Independent Researcher <sup>4</sup>UC San Diego

## Abstract

High throughput serving of large language models (LLMs) requires batching sufficiently many requests at a time. However, existing systems struggle because the key-value cache (KV cache) memory for each request is huge and grows and shrinks dynamically. When managed inefficiently, this memory can be significantly wasted by fragmentation and redundant duplication, limiting the batch size. To address this problem, we propose PagedAttention, an attention algorithm inspired by the classical virtual memory and paging techniques in operating systems. On top of it, we build vLLM, an LLM serving system that achieves (1) near-zero waste in KV cache memory and (2) flexible sharing of KV cache within and across requests to further reduce memory usage. Our evaluations show that vLLM improves the throughput of popular LLMs by 2-4× with the same level of latency compared to the state-of-the-art systems, such as FasterTransformer and Orca. The improvement is more pronounced with longer sequences, larger models, and more complex decoding algorithms. vLLM’s source code is publicly available at <https://github.com/vllm-project/vllm>.

## 1 Introduction

The emergence of large language models (LLMs) like GPT [5, 37] and PaLM [9] have enabled new applications such as programming assistants [6, 18] and universal chatbots [19, 35] that are starting to profoundly impact our work and daily routines. Many cloud companies [34, 44] are racing to provide these applications as hosted services. However, running these applications is very expensive, requiring a large number of hardware accelerators such as GPUs. According to recent estimates, processing an LLM request can be 10× more expensive than a traditional keyword query [43]. Given these high costs, increasing the throughput—and hence reducing

---

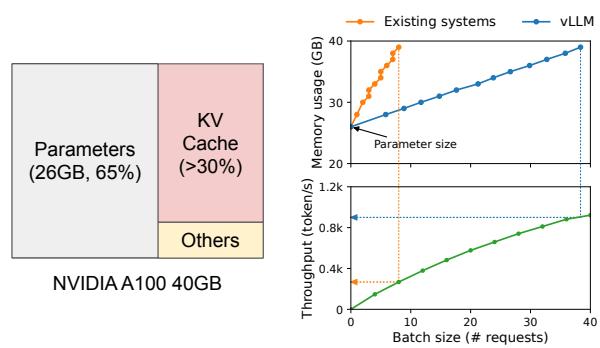
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SOSP ’23, October 23–26, 2023, Koblenz, Germany

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0229-7/23/10.

<https://doi.org/10.1145/3600006.3613165>



**Figure 1.** Left: Memory layout when serving an LLM with 13B parameters on NVIDIA A100. The parameters (gray) persist in GPU memory throughout serving. The memory for the KV cache (red) is (de)allocated per serving request. A small amount of memory (yellow) is used ephemerally for activation. Right: vLLM smooths out the rapid growth curve of KV cache memory seen in existing systems [31, 60], leading to a notable boost in serving throughput.

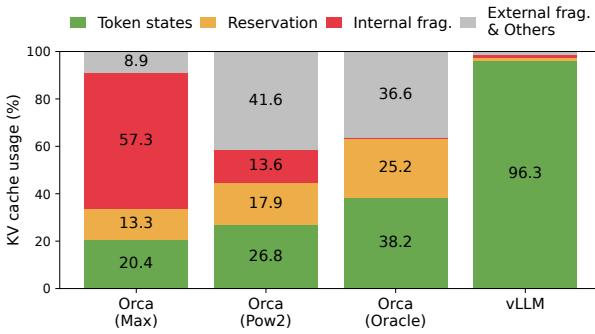
激活值占的显存比较少，权重和kv cache占大部分

the cost per request—of *LLM serving* systems is becoming more important.

At the core of LLMs lies an *autoregressive* Transformer model [53]. This model generates words (tokens), *one at a time*, based on the input (prompt) and the previous sequence of the output’s tokens it has generated so far. For each request, this expensive process is repeated until the model outputs a termination token. This sequential generation process makes the workload *memory-bound*, underutilizing the computation power of GPUs and limiting the serving throughput.

Improving the throughput is possible by batching multiple requests together. However, to process many requests in a batch, the memory space for each request should be efficiently managed. For example, Fig. 1 (left) illustrates the memory distribution for a 13B-parameter LLM on an NVIDIA A100 GPU with 40GB RAM. Approximately 65% of the memory is allocated for the model weights, which remain static during serving. Close to 30% of the memory is used to store the dynamic states of the requests. For Transformers, these states consist of the key and value tensors associated with the attention mechanism, commonly referred to as *KV cache* [41], which represent the context from earlier tokens to generate new output tokens in sequence. The remaining small

\*Equal contribution.



**Figure 2.** Average percentage of memory wastes in different LLM serving systems during the experiment in §6.2.

percentage of memory is used for other data, including activations – the ephemeral tensors created when evaluating the LLM. Since the model weights are constant and the activations only occupy a small fraction of the GPU memory, the way the KV cache is managed is critical in determining the maximum batch size. When managed inefficiently, the KV cache memory can significantly limit the batch size and consequently the throughput of the LLM, as illustrated in Fig. 1 (right).

In this paper, we observe that existing LLM serving systems [31, 60] fall short of managing the KV cache memory efficiently. This is mainly because they store the KV cache of a request in contiguous memory space, as most deep learning frameworks [33, 39] require tensors to be stored in contiguous memory. However, unlike the tensors in the traditional deep learning workloads, the KV cache has unique characteristics: it dynamically grows and shrinks over time as the model generates new tokens, and its lifetime and length are not known a priori. These characteristics make the existing systems’ approach significantly inefficient in two ways:

First, the existing systems [31, 60] suffer from internal and external memory fragmentation. To store the KV cache of a request in contiguous space, they pre-allocate a contiguous chunk of memory with the request’s maximum length (e.g., 2048 tokens). This can result in severe internal fragmentation, since the request’s actual length can be much shorter than its maximum length (e.g., Fig. 11). Moreover, even if the actual length is known a priori, the pre-allocation is still inefficient: As the entire chunk is reserved during the request’s lifetime, other shorter requests cannot utilize any part of the chunk that is currently unused. Besides, external memory fragmentation can also be significant, since the pre-allocated size can be different for each request. Indeed, our profiling results in Fig. 2 show that only 20.4% - 38.2% of the KV cache memory is used to store the actual token states in the existing systems.

Second, the existing systems cannot exploit the opportunities for memory sharing. LLM services often use advanced

激活值张量  
是临时的

decoding algorithms, such as parallel sampling and beam search, that generate multiple outputs per request. In these scenarios, the request consists of multiple sequences that can partially share their KV cache. However, memory sharing is not possible in the existing systems because the KV cache of the sequences is stored in separate contiguous spaces.

To address the above limitations, we propose *PagedAttention*, an attention algorithm inspired by the operating system’s (OS) solution to memory fragmentation and sharing: virtual memory with paging. PagedAttention divides the request’s KV cache into blocks, each of which can contain the attention keys and values of a fixed number of tokens. In PagedAttention, the blocks for the KV cache are not necessarily stored in contiguous space. Therefore, we can manage the KV cache in a more flexible way as in OS’s virtual memory: one can think of blocks as pages, tokens as bytes, and requests as processes. This design alleviates internal fragmentation by using relatively small blocks and allocating them on demand. Moreover, it eliminates external fragmentation as all blocks have the same size. Finally, it enables memory sharing at the granularity of a block, across the different sequences associated with the same request or even across the different requests.

In this work, we build vLLM, a high-throughput distributed LLM serving engine on top of PagedAttention that achieves near-zero waste in KV cache memory. vLLM uses block-level memory management and preemptive request scheduling that are co-designed with PagedAttention. vLLM supports popular LLMs such as GPT [5], OPT [62], and LLaMA [52] with varying sizes, including the ones exceeding the memory capacity of a single GPU. Our evaluations on various models and workloads show that vLLM improves the LLM serving throughput by 2-4× compared to the state-of-the-art systems [31, 60], without affecting the model accuracy at all. The improvements are more pronounced with longer sequences, larger models, and more complex decoding algorithms (§4.3). In summary, we make the following contributions:

- We identify the challenges in memory allocation in serving LLMs and quantify their impact on serving performance.
- We propose PagedAttention, an attention algorithm that operates on KV cache stored in non-contiguous paged memory, which is inspired by the virtual memory and paging in OS.
- We design and implement vLLM, a distributed LLM serving engine built on top of PagedAttention.
- We evaluate vLLM on various scenarios and demonstrate that it substantially outperforms the previous state-of-the-art solutions such as FasterTransformer [31] and Orca [60].

## 2 Background

In this section, we describe the generation and serving procedures of typical LLMs and the iteration-level scheduling used in LLM serving.

## 2.1 Transformer-Based Large Language Models

The task of language modeling is to model the probability of a list of tokens  $(x_1, \dots, x_n)$ . Since language has a natural sequential ordering, it is common to factorize the joint probability over the whole sequence as the product of conditional probabilities (a.k.a. *autoregressive decomposition* [3]):

$$P(x) = P(x_1) \cdot P(x_2 | x_1) \cdots P(x_n | x_1, \dots, x_{n-1}). \quad (1)$$

事實上的

Transformers [53] have become the de facto standard architecture for modeling the probability above at a large scale. The most important component of a Transformer-based language model is its *self-attention* layers. For an input hidden state sequence  $(x_1, \dots, x_n) \in \mathbb{R}^{n \times d}$ , a self-attention layer first applies linear transformations on each position  $i$  to get the query, key, and value vectors:

$$q_i = W_q x_i, k_i = W_k x_i, v_i = W_v x_i. \quad (2)$$

Then, the self-attention layer computes the attention score  $a_{ij}$  by multiplying the query vector at one position with all the key vectors before it and compute the output  $o_i$  as the weighted average over the value vectors:

$$a_{ij} = \frac{\exp(q_i^\top k_j / \sqrt{d})}{\sum_{t=1}^i \exp(q_i^\top k_t / \sqrt{d})}, o_i = \sum_{j=1}^i a_{ij} v_j. \quad (3)$$

Besides the computation in Eq. 4, all other components in the Transformer model, including the embedding layer, feed-forward layer, layer normalization [2], residual connection [22], output logit computation, and the query, key, and value transformation in Eq. 2, are all applied independently position-wise in a form of  $y_i = f(x_i)$ .

## 2.2 LLM Service & Autoregressive Generation

Once trained, LLMs are often deployed as a conditional generation service (e.g., completion API [34] or chatbot [19, 35]). A request to an LLM service provides a list of *input prompt* tokens  $(x_1, \dots, x_n)$ , and the LLM service generates a list of output tokens  $(x_{n+1}, \dots, x_{n+T})$  according to Eq. 1. We refer to the concatenation of the prompt and output lists as *sequence*.

Due to the decomposition in Eq. 1, the LLM can only sample and generate new tokens one by one, and the generation process of each new token depends on all the previous tokens in that sequence, specifically their key and value vectors. In this sequential generation process, the key and value vectors of existing tokens are often cached for generating future tokens, known as *KV cache*. Note that the KV cache of one token depends on all its previous tokens. This means that the KV cache of the same token appearing at different positions in a sequence will be different.

Given a request prompt, the generation computation in the LLM service can be decomposed into two phases:

**The prompt phase** takes the whole user prompt  $(x_1, \dots, x_n)$  as input and computes the probability of the first new token  $P(x_{n+1} | x_1, \dots, x_n)$ . During this process, also generates the key vectors  $k_1, \dots, k_n$  and value vectors  $v_1, \dots, v_n$ . Since prompt tokens  $x_1, \dots, x_n$  are all known, the computation of the prompt phase can be parallelized using matrix-matrix multiplication operations. Therefore, this phase can efficiently use the parallelism inherent in GPUs.

**The autoregressive generation phase** generates the remaining new tokens sequentially. At iteration  $t$ , the model takes one token  $x_{n+t}$  as input and computes the probability  $P(x_{n+t+1} | x_1, \dots, x_{n+t})$  with the key vectors  $k_1, \dots, k_{n+t}$  and value vectors  $v_1, \dots, v_{n+t}$ . Note that the key and value vectors at positions 1 to  $n+t-1$  are cached at previous iterations, only the new key and value vector  $k_{n+t}$  and  $v_{n+t}$  are computed at this iteration. This phase completes either when the sequence reaches a maximum length (specified by users or limited by LLMs) or when an end-of-sequence (`<eos>`) token is emitted. The computation at different iterations cannot be parallelized due to the data dependency and often uses matrix-vector multiplication, which is less efficient. As a result, this phase severely underutilizes GPU computation and becomes memory-bound, being responsible for most portion of the latency of a single request.

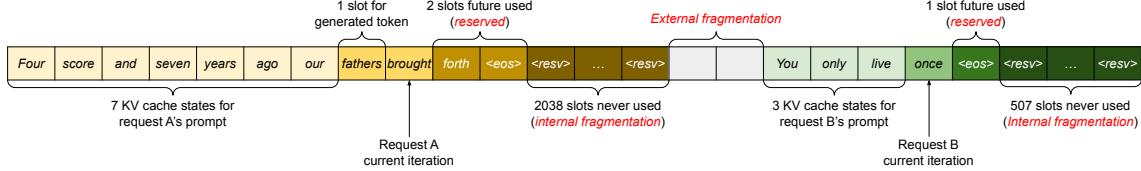
GEMV

## 2.3 Batching Techniques for LLMs

The compute utilization in serving LLMs can be improved by batching multiple requests. Because the requests share the same model weights, the overhead of moving weights is amortized across the requests in a batch, and can be overwhelmed by the computational overhead when the batch size is sufficiently large. However, batching the requests to an LLM service is non-trivial for two reasons. First, the requests may arrive at different times. A naive batching strategy would either make earlier requests wait for later ones or delay the incoming requests until earlier ones finish, leading to significant queueing delays. Second, the requests may have vastly different input and output lengths (Fig. 11). A straightforward batching technique would pad the inputs and outputs of the requests to equalize their lengths, wasting GPU computation and memory.

To address this problem, fine-grained batching mechanisms, such as cellular batching [16] and iteration-level scheduling [60], have been proposed. Unlike traditional methods that work at the request level, these techniques operate at the iteration level. After each iteration, completed requests are removed from the batch, and new ones are added. Therefore, a new request can be processed after waiting for a single iteration, not waiting for the entire batch to complete. Moreover, with special GPU kernels, these techniques eliminate the need to pad the inputs and outputs. By reducing the queueing delay and the inefficiencies from padding, the fine-grained batching mechanisms significantly increase the throughput of LLM serving.

权重加载的耗时可以被计算延迟隐藏，当bs比较大的时候



**Figure 3.** KV cache memory management in existing systems. Three types of memory wastes – **reserved**, **internal fragmentation**, and **external fragmentation** – exist that prevent other requests from fitting into the memory. The token in each memory slot represents its KV cache. Note the same tokens can have different KV cache when at different positions.

### 3 Memory Challenges in LLM Serving

Although fine-grained batching reduces the waste of computing and enables requests to be batched in a more flexible way, the number of requests that can be batched together is still constrained by GPU memory capacity, particularly the space allocated to store the KV cache. In other words, the serving system’s throughput is *memory-bound*. Overcoming this memory-bound requires addressing the following challenges in the memory management:

**Large KV cache.** The KV Cache size grows quickly with the number of requests. As an example, for the 13B parameter OPT model [62], the KV cache of a single token demands 800 KB of space, calculated as 2 (key and value vectors)  $\times$  5120 (hidden state size)  $\times$  40 (number of layers)  $\times$  2 (bytes per FP16). Since OPT can generate sequences up to 2048 tokens, the memory required to store the KV cache of one request can be as much as 1.6 GB. Concurrent GPUs have memory capacities in the tens of GBs. Even if all available memory was allocated to KV cache, only a few tens of requests could be accommodated. Moreover, inefficient memory management can further decrease the batch size, as shown in Fig. 2. Additionally, given the current trends, the GPU’s computation speed grows faster than the memory capacity [17]. For example, from NVIDIA A100 to H100, The FLOPS increases by more than 2x, but the GPU memory stays at 80GB maximum. Therefore, we believe the memory will become an increasingly significant bottleneck.

**Complex decoding algorithms.** LLM services offer a range of decoding algorithms for users to select from, each with varying implications for memory management complexity. For example, when users request multiple random samples from a single input prompt, a typical use case in program suggestion [18], the KV cache of the prompt part, which accounts for 12% of the total KV cache memory in our experiment (§6.3), can be shared to minimize memory usage. On the other hand, the KV cache during the autoregressive generation phase should remain unshared due to the different sample results and their dependence on context and position. The extent of KV cache sharing depends on the specific decoding algorithm employed. In more sophisticated algorithms like beam search [49], different request beams can share larger portions (up to 55% memory saving, see

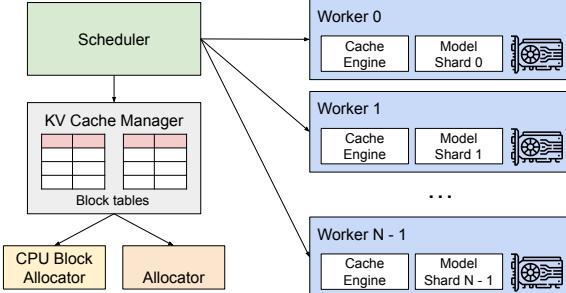
§6.3) of their KV cache, and the sharing pattern evolves as the decoding process advances.

**Scheduling for unknown input & output lengths.** The requests to an LLM service exhibit variability in their input and output lengths. This requires the memory management system to accommodate a wide range of prompt lengths. In addition, as the output length of a request grows at decoding, the memory required for its KV cache also expands and may exhaust available memory for incoming requests or ongoing generation for existing prompts. The system needs to make scheduling decisions, such as deleting or swapping out the KV cache of some requests from GPU memory.

#### 3.1 Memory Management in Existing Systems

Since most operators in current deep learning frameworks [33, 39] require tensors to be stored in contiguous memory, previous LLM serving systems [31, 60] also store the KV cache of one request as a contiguous tensor across the different positions. Due to the unpredictable output lengths from the LLM, they statically allocate a chunk of memory for a request based on the request’s maximum possible sequence length, irrespective of the actual input or eventual output length of the request.

Fig. 3 illustrates two requests: request A with 2048 maximum possible sequence length and request B with a maximum of 512. The chunk pre-allocation scheme in existing systems has three primary sources of memory wastes: reserved slots for future tokens, internal fragmentation due to over-provisioning for potential maximum sequence lengths, and external fragmentation from the memory allocator like the buddy allocator. The external fragmentation will never be used for generated tokens, which is known before serving a request. Internal fragmentation also remains unused, but this is only realized after a request has finished sampling. They are both pure memory waste. Although the reserved memory is eventually used, reserving this space for the entire request’s duration, especially when the reserved space is large, occupies the space that could otherwise be used to process other requests. We visualize the average percentage of memory wastes in our experiments in Fig. 2, revealing that the actual effective memory in previous systems can be as low as 20.4%.



**Figure 4.** vLLM system overview.

Although compaction [54] has been proposed as a potential solution to fragmentation, performing compaction in a performance-sensitive LLM serving system is impractical due to the massive KV cache. Even with compaction, the pre-allocated chunk space for each request prevents memory sharing specific to decoding algorithms in existing memory management systems.

## 4 Method

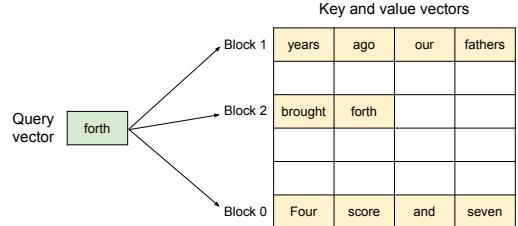
In this work, we develop a new attention algorithm, *PagedAttention*, and build an LLM serving engine, *vLLM*, to tackle the challenges outlined in §3. The architecture of vLLM is shown in Fig. 4. vLLM adopts a centralized scheduler to coordinate the execution of distributed GPU workers. The *KV cache manager* effectively manages the KV cache in a paged fashion, enabled by PagedAttention. Specifically, the KV cache manager manages the physical KV cache memory on the GPU workers through the instructions sent by the centralized scheduler.

Next, We describe the PagedAttention algorithm in §4.1. With that, we show the design of the KV cache manager in §4.2 and how it facilitates PagedAttention in §4.3, respectively. Then, we show how this design facilitates effective memory management for various decoding methods (§4.4) and handles the variable length input and output sequences (§4.5). Finally, we show how the system design of vLLM works in a distributed setting (§4.6).

### 4.1 PagedAttention

To address the memory challenges in §3, we introduce *PagedAttention*, an attention algorithm inspired by the classic idea of *paging* [25] in operating systems. Unlike the traditional attention algorithms, PagedAttention allows storing continuous keys and values in non-contiguous memory space. Specifically, PagedAttention partitions the KV cache of each sequence into KV blocks. Each block contains the key and value vectors for a fixed number of tokens,<sup>1</sup> which we denote as *KV*

<sup>1</sup>In Transformer, each token has a set of key and value vectors across layers and attention heads within a layer. All the key and value vectors can be managed together within a single KV block, or the key and value vectors at different heads and layers can each have a separate block and be managed in separate block tables. The two designs have no performance difference and we choose the second one for easy implementation.



**Figure 5.** Illustration of the PagedAttention algorithm, where the attention key and values vectors are stored as non-contiguous blocks in the memory.

*block size (B)*. Denote the key block  $K_j = (k_{(j-1)B+1}, \dots, k_{jB})$  and value block  $V_j = (v_{(j-1)B+1}, \dots, v_{jB})$ . The attention computation in Eq. 4 can be transformed into the following block-wise computation:

$$A_{ij} = \frac{\exp(q_i^\top K_j / \sqrt{d})}{\sum_{t=1}^{\lceil i/B \rceil} \exp(q_i^\top K_t 1 / \sqrt{d})}, \quad o_i = \sum_{j=1}^{\lceil i/B \rceil} V_j A_{ij}^\top, \quad (4)$$

where  $A_{ij} = (a_{i,(j-1)B+1}, \dots, a_{i,jB})$  is the row vector of attention score on  $j$ -th KV block.

During the attention computation, the PagedAttention kernel identifies and fetches different KV blocks separately. We show an example of PagedAttention in Fig. 5: The key and value vectors are spread across three blocks, and the three blocks are not contiguous on the physical memory. At each time, the kernel multiplies the query vector  $q_i$  of the query token (“forth”) and the key vectors  $K_j$  in a block (e.g., key vectors of “Four score and seven” for block 0) to compute the attention score  $A_{ij}$ , and later multiplies  $A_{ij}$  with the value vectors  $V_j$  in a block to derive the final attention output  $o_i$ .

In summary, the PagedAttention algorithm allows the KV blocks to be stored in non-contiguous physical memory, which enables more flexible paged memory management in vLLM.

### 4.2 KV Cache Manager

The key idea behind vLLM’s memory manager is analogous to the *virtual memory* [25] in operating systems. OS partitions memory into fixed-sized *pages* and maps user programs’ logical pages to physical pages. Contiguous logical pages can correspond to non-contiguous physical memory pages, allowing user programs to access memory as though it were contiguous. Moreover, physical memory space needs not to be fully reserved in advance, enabling the OS to dynamically allocate physical pages as needed. vLLM uses the ideas behind virtual memory to manage the KV cache in an LLM service. Enabled by PagedAttention, we organize the KV cache as fixed-size KV blocks, like pages in virtual memory.

A request’s KV cache is represented as a series of logical KV blocks, filled from left to right as new tokens and their KV cache are generated. The last KV block’s unfilled positions are reserved for future generations. On GPU workers, a *block engine* allocates a contiguous chunk of GPU DRAM and

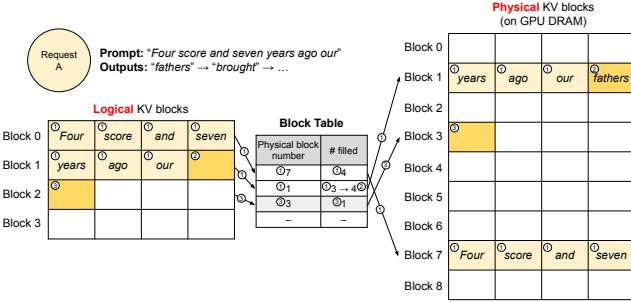


Figure 6. Block table translation in vLLM.

divides it into *physical KV blocks* (this is also done on CPU RAM for swapping; see §4.5). The *KV block manager* also maintains *block tables*—the mapping between *logical* and *physical KV blocks* of each request. Each block table entry records the corresponding physical blocks of a logical block and the number of filled positions. Separating logical and physical KV blocks allows vLLM to dynamically grow the KV cache memory without reserving it for all positions in advance, which eliminates most memory waste in existing systems, as in Fig. 2.

#### 4.3 Decoding with PagedAttention and vLLM

Next, we walk through an example, as in Fig. 6, to demonstrate how vLLM executes PagedAttention and manages the memory during the decoding process of a single input sequence: ① As in OS’s virtual memory, vLLM does not require reserving the memory for the maximum possible generated sequence length initially. Instead, it reserves only the necessary KV blocks to accommodate the KV cache generated during prompt computation. In this case, The prompt has 7 tokens, so vLLM maps the first 2 logical KV blocks (0 and 1) to 2 physical KV blocks (7 and 1, respectively). In the *prefill* step, vLLM generates the KV cache of the prompts and the first output token with a *conventional* self-attention algorithm (e.g., [13]). vLLM then stores the KV cache of the first 4 tokens in logical block 0 and the following 3 tokens in logical block 1. The remaining slot is reserved for the subsequent autoregressive generation phase. ② In the first autoregressive decoding step, vLLM generates the new token with the PagedAttention algorithm on physical blocks 7 and 1. Since one slot remains available in the last logical block, the newly generated KV cache is stored there, and the block table’s #filled record is updated. ③ At the second decoding step, as the last logical block is full, vLLM stores the newly generated KV cache in a new logical block; vLLM allocates a new physical block (physical block 3) for it and stores this mapping in the block table.

Globally, for each decoding iteration, vLLM first selects a set of candidate sequences for batching (more in §4.5), and allocates the physical blocks for the newly required logical blocks. Then, vLLM concatenates all the input tokens of the current iteration (i.e., all tokens for prompt phase

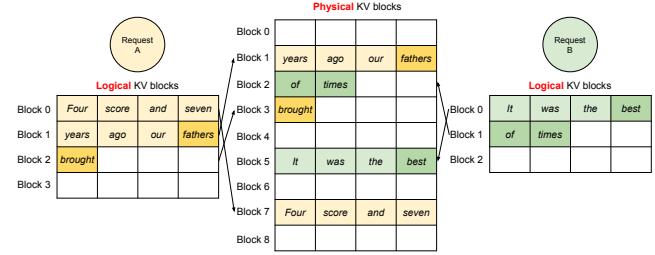


Figure 7. Storing the KV cache of two requests at the same time in vLLM.

将所有seq拼接成一个大seq输入LLM进行推理

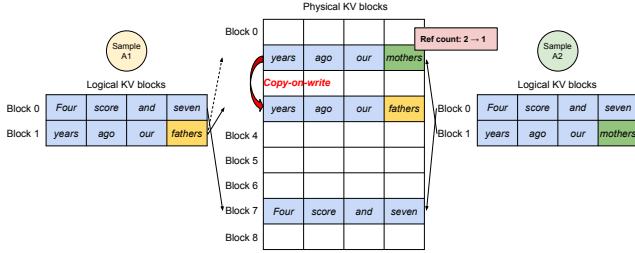
requests and the latest tokens for generation phase requests) as one sequence and feeds it into the LLM. During LLM’s computation, vLLM uses the PagedAttention kernel to access the previous KV cache stored in the form of logical KV blocks and saves the newly generated KV cache into the physical KV blocks. Storing multiple tokens within a KV block (block size > 1) enables the PagedAttention kernel to process the KV cache across more positions in parallel, thus increasing the hardware utilization and reducing latency. However, a larger block size also increases memory fragmentation. We study the effect of block size in §7.2.

Again, vLLM dynamically assigns new physical blocks to logical blocks as more tokens and their KV cache are generated. As all the blocks are filled from left to right and a new physical block is only allocated when all previous blocks are full, vLLM limits all the memory wastes for a request within one block, so it can effectively utilize all the memory, as shown in Fig. 2. This allows more requests to fit into memory for batching—hence improving the throughput. Once a request finishes its generation, its KV blocks can be freed to store the KV cache of other requests. In Fig. 7, we show an example of vLLM managing the memory for two sequences. The logical blocks of the two sequences are mapped to different physical blocks within the space reserved by the block engine in GPU workers. The neighboring logical blocks of both sequences do not need to be contiguous in physical GPU memory and the space of physical blocks can be effectively utilized by both sequences.

#### 4.4 Application to Other Decoding Scenarios

§4.3 shows how PagedAttention and vLLM handle basic decoding algorithms, such as greedy decoding and sampling, that take one user prompt as input and generate a single output sequence. In many successful LLM applications [18, 34], an LLM service must offer more complex decoding scenarios that exhibit complex accessing patterns and more opportunities for memory sharing. We show the general applicability of vLLM on them in this section.

**Parallel sampling.** In LLM-based program assistants [6, 18], an LLM generates multiple sampled outputs for a single input prompt; users can choose a favorite output from various candidates. So far we have implicitly assumed that a request



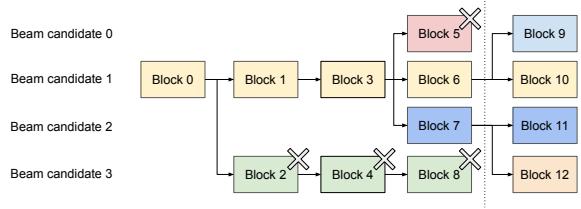
**Figure 8.** Parallel sampling example.

**generates a single sequence.** In the remainder of this paper, we **assume the more general case in which a request generates multiple sequences.** In parallel sampling, one request includes multiple samples **sharing the same input prompt**, allowing the KV cache of the prompt to be shared as well. Via its PagedAttention and paged memory management, vLLM can realize this sharing easily and save memory.

Fig. 8 shows an example of parallel decoding for two outputs. Since both outputs **share the same prompt**, we only reserve space for one copy of the prompt’s state at the prompt phase; **the logical blocks for the prompts of both sequences are mapped to the same physical blocks**: the logical block 0 and 1 of both sequences are mapped to physical blocks 7 and 1, respectively. Since a single physical block can be mapped to multiple logical blocks, we introduce a **reference count** for each **physical block**. In this case, the reference counts for physical blocks 7 and 1 are **both 2**. At the generation phase, the **two outputs sample different output tokens and need separate storage for KV cache**. vLLM implements a **copy-on-write** mechanism at the **block granularity** for the physical blocks that need modification by multiple sequences, similar to the copy-on-write technique in OS virtual memory (e.g., when forking a process). Specifically, in Fig. 8, when sample A1 needs to write to its last logical block (logical block 1), vLLM recognizes that the reference count of the corresponding physical block (physical block 1) is greater than 1; it **allocates a new physical block (physical block 3)**, **instructs the block engine to copy the information from physical block 1**, and **decreases the reference count to 1**. Next, when sample A2 writes to physical block 1, **the reference count is already reduced to 1**; thus A2 directly writes its newly generated KV cache to physical block 1.

In summary, vLLM enables the **sharing** of most of the space used to store the **prompts’ KV cache** across multiple output samples, **with the exception of the final logical block**, which is managed by a **copy-on-write** mechanism. By sharing physical blocks across multiple samples, memory usage can be greatly reduced, especially for **long input prompts**.

**Beam search.** In LLM tasks like machine translation [59], the users expect the top- $k$  most appropriate translations output by the LLM. Beam search [49] is widely used to decode the most probable output sequence from an LLM, as it mitigates the computational complexity of fully traversing the



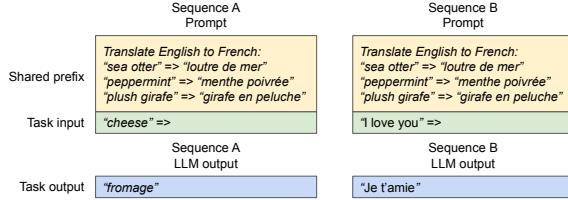
**Figure 9.** Beam search example.

sample space. The algorithm relies on the **beam width** parameter  $k$ , which determines the number of top candidates retained at every step. During decoding, beam search expands each candidate sequence in the beam by considering all possible tokens, computes their respective probabilities using the LLM, and retains the top- $k$  most probable sequences out of  $k \cdot |V|$  candidates, where  $|V|$  is the vocabulary size.

Unlike parallel decoding, **beam search facilities sharing not only the initial prompt blocks but also other blocks across different candidates**, and the sharing patterns dynamically change as the decoding process advances, similar to the process tree in the OS created by compound forks. Fig. 9 shows how vLLM manages the KV blocks for a beam search example with  $k = 4$ . Prior to the iteration illustrated as the dotted line, each candidate sequence has used 4 full logical blocks. All beam candidates **share the first block 0** (i.e., prompt). Candidate 3 digresses from others from the second block. Candidates 0-2 share the first 3 blocks and diverge at the fourth block. At subsequent iterations, **the top-4 probable candidates all originate from candidates 1 and 2**. As the original candidates 0 and 3 are no longer among the top candidates, **their logical blocks are freed, and the reference counts of corresponding physical blocks are reduced**. vLLM frees all physical blocks whose reference counts reach 0 (blocks 2, 4, 5, 8). Then, vLLM allocates new physical blocks (blocks 9-12) to store the new KV cache from the new candidates. Now, **all candidates share blocks 0, 1, 3; candidates 0 and 1 share block 6, and candidates 2 and 3 further share block 7**.

Previous LLM serving systems require frequent memory copies of the KV cache across the beam candidates. For example, in the case shown in Fig. 9, after the dotted line, candidate 3 would need to copy a large portion of candidate 2’s KV cache to continue generation. This frequent memory copy overhead is significantly reduced by vLLM’s physical block sharing. In vLLM, most blocks of different beam candidates can be shared. The copy-on-write mechanism is applied only when the newly generated tokens are within an old shared block, as in parallel decoding. This involves only copying one block of data.

**Shared prefix.** Commonly, the LLM user provides a (long) description of the task including instructions and example inputs and outputs, also known as *system prompt* [36]. The description is concatenated with the actual task input to form the prompt of the request. The LLM generates outputs based



**Figure 10.** Shared prompt example for machine translation. The examples are adopted from [5].

on the full prompt. Fig. 10 shows an example. Moreover, the shared prefix can be further tuned, via prompt engineering, to improve the accuracy of the downstream tasks [26, 27].

For this type of application, many user prompts share a prefix, thus the LLM service provider can store the KV cache of the prefix in advance to reduce the redundant computation spent on the prefix. In vLLM, this can be conveniently achieved by reserving a set of physical blocks for a set of predefined shared prefixes by the LLM service provider, as how OS handles shared library across processes. A user input prompt with the shared prefix can simply map its logical blocks to the cached physical blocks (with the last block marked copy-on-write). The prompt phase computation only needs to execute on the user’s task input.

**Mixed decoding methods.** The decoding methods discussed earlier exhibit diverse memory sharing and accessing patterns. Nonetheless, vLLM facilitates the simultaneous processing of requests with different decoding preferences, which existing systems cannot efficiently do. This is because vLLM conceals the complex memory sharing between different sequences via a common mapping layer that translates logical blocks to physical blocks. The LLM and its execution kernel only see a list of physical block IDs for each sequence and do not need to handle sharing patterns across sequences. Compared to existing systems, this approach broadens the batching opportunities for requests with different sampling requirements, ultimately increasing the system’s overall throughput.

#### 4.5 Scheduling and Preemption

When the request traffic surpasses the system’s capacity, vLLM must prioritize a subset of requests. In vLLM, we adopt the first-come-first-serve (FCFS) scheduling policy for all requests, ensuring fairness and preventing starvation. When vLLM needs to preempt requests, it ensures that the earliest arrived requests are served first and the latest requests are preempted first.

LLM services face a unique challenge: the input prompts for an LLM can vary significantly in length, and the resulting output lengths are not known a priori, contingent on both the input prompt and the model. As the number of requests and their outputs grow, vLLM can run out of the GPU’s physical blocks to store the newly generated KV cache. There are two classic questions that vLLM needs to answer in this

context: (1) Which blocks should it evict? (2) How to recover evicted blocks if needed again?

启发式 Typically, eviction policies use heuristics to predict which block will be accessed furthest in the future and evict that block. Since in our case we know that all blocks of a sequence are accessed together, we implement an all-or-nothing eviction policy, i.e., either evict all or none of the blocks of a sequence. Furthermore, multiple sequences within one request (e.g., beam candidates in one beam search request) are gang-scheduled as a sequence group. The sequences within one sequence group are always preempted or rescheduled together due to potential memory sharing across those sequences. To answer the second question of how to recover an evicted block, we consider two techniques:

**Swapping.** This is the classic technique used by most virtual memory implementations which copy the evicted pages to a swap space on the disk. In our case, we copy evicted blocks to the CPU memory. As shown in Fig. 4, besides the GPU block allocator, vLLM includes a CPU block allocator to manage the physical blocks swapped to CPU RAM. When vLLM exhausts free physical blocks for new tokens, it selects a set of sequences to evict and transfer their KV cache to the CPU. Once it preempts a sequence and evicts its blocks, vLLM stops accepting new requests until all preempted sequences are completed. Once a request completes, its blocks are freed from memory, and the blocks of a preempted sequence are brought back in to continue the processing of that sequence. Note that with this design, the number of blocks swapped to the CPU RAM never exceeds the number of total physical blocks in the GPU RAM, so the swap space on the CPU RAM is bounded by the GPU memory allocated for the KV cache.

**Recomputation.** In this case, we simply recompute the KV cache when the preempted sequences are rescheduled. Note that recomputation latency can be significantly lower than the original latency, as the tokens generated at decoding can be concatenated with the original user prompt as a new prompt—their KV cache at all positions can be generated in one prompt phase iteration.

The performances of swapping and recomputation depend on the bandwidth between CPU RAM and GPU memory and the computation power of the GPU. We examine the speeds of swapping and recomputation in §7.3.

#### 4.6 Distributed Execution

Many LLMs have parameter sizes exceeding the capacity of a single GPU [5, 9]. Therefore, it is necessary to partition them across distributed GPUs and execute them in a model parallel fashion [28, 63]. This calls for a memory manager capable of handling distributed memory. vLLM is effective in distributed settings by supporting the widely used Megatron-LM style tensor model parallelism strategy on Transformers [47]. This strategy adheres to an SPMD (Single Program Multiple Data) execution schedule, wherein the linear layers are partitioned

**Table 1.** Model sizes and server configurations.

Model size	13B	66B	175B
GPUs	A100	4×A100	8×A100-80GB
Total GPU memory	40 GB	160 GB	640 GB
Parameter size	26 GB	132 GB	346 GB
Memory for KV cache	12 GB	21 GB	264 GB
Max. # KV cache slots	15.7K	9.7K	60.1K

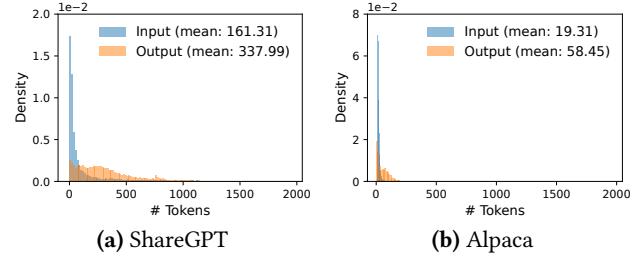
to perform block-wise matrix multiplication, and the GPUs constantly synchronize intermediate results via an all-reduce operation. Specifically, the attention operator is split on the attention head dimension, each SPMD process takes care of a subset of attention heads in multi-head attention.

We observe that even with model parallel execution, each model shard still processes the same set of input tokens, thus requiring the KV Cache for the same positions. Therefore, vLLM features a single KV cache manager within the centralized scheduler, as in Fig. 4. Different GPU workers share the manager, as well as the mapping from logical blocks to physical blocks. This common mapping allows GPU workers to execute the model with the physical blocks provided by the scheduler for each input request. Although each GPU worker has the same physical block IDs, a worker only stores a portion of the KV cache for its corresponding attention heads.

In each step, the scheduler first prepares the message with input token IDs for each request in the batch, as well as the block table for each request. Next, the scheduler broadcasts this control message to the GPU workers. Then, the GPU workers start to execute the model with the input token IDs. In the attention layers, the GPU workers read the KV cache according to the block table in the control message. During execution, the GPU workers synchronize the intermediate results with the all-reduce communication primitive without the coordination of the scheduler, as in [47]. In the end, the GPU workers send the sampled tokens of this iteration back to the scheduler. In summary, GPU workers do not need to synchronize on memory management as they only need to receive all the memory management information at the beginning of each decoding iteration along with the step inputs.

## 5 Implementation

vLLM is an end-to-end serving system with a FastAPI [15] frontend and a GPU-based inference engine. The frontend extends the OpenAI API [34] interface, allowing users to customize sampling parameters for each request, such as the maximum sequence length and the beam width  $k$ . The vLLM engine is written in 8.5K lines of Python and 2K lines of C++/CUDA code. We develop control-related components including the scheduler and the block manager in Python while developing custom CUDA kernels for key operations such as PagedAttention. For the model executor, we implement popular LLMs such as GPT [5], OPT [62], and LLaMA [52] using



**Figure 11.** Input and output length distributions of the (a) ShareGPT and (b) Alpaca datasets.

PyTorch [39] and Transformers [58]. We use NCCL [32] for tensor communication across the distributed GPU workers.

### 5.1 Kernel-level Optimization

Since PagedAttention introduces memory access patterns that are not efficiently supported by existing systems, we develop several GPU kernels for optimizing it. (1) *Fused reshape and block write*. In every Transformer layer, the new KV cache are split into blocks, reshaped to a memory layout optimized for block read, then saved at positions specified by the block table. To minimize kernel launch overheads, we fuse them into a single kernel. (2) *Fusing block read and attention*. We adapt the attention kernel in FasterTransformer [31] to read KV cache according to the block table and perform attention operations on the fly. To ensure coalesced memory access, we assign a GPU warp to read each block. Moreover, we add support for variable sequence lengths within a request batch. (3) *Fused block copy*. Block copy operations, issued by the copy-on-write mechanism, may operate on discontinuous blocks. This can lead to numerous invocations of small data movements if we use the cudaMemcpyAsync API. To mitigate the overhead, we implement a kernel that batches the copy operations for different blocks into a single kernel launch.

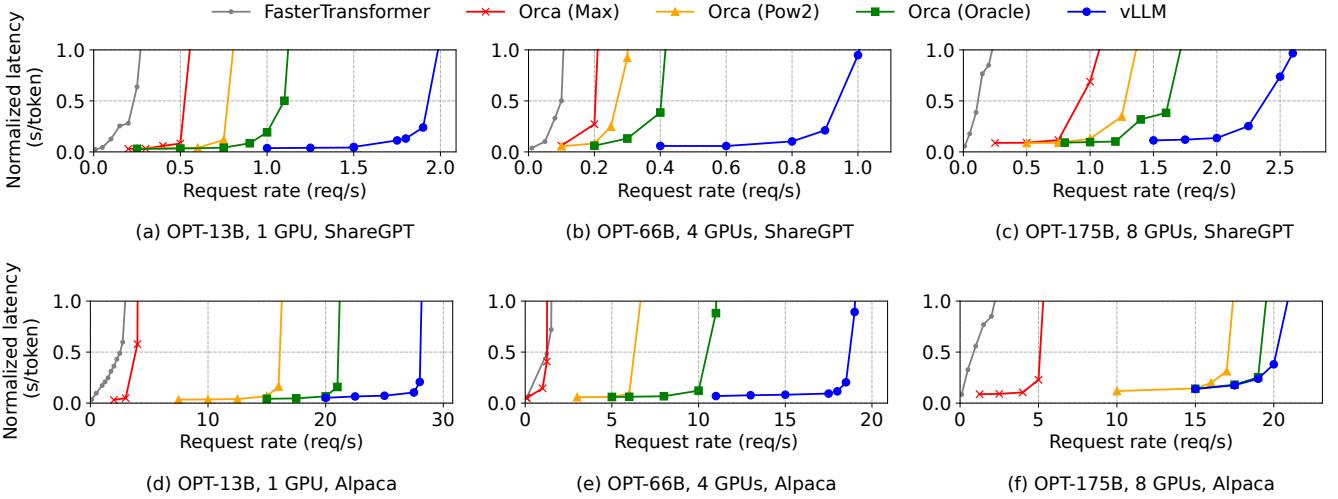
### 5.2 Supporting Various Decoding Algorithms

vLLM implements various decoding algorithms using three key methods: fork, append, and free. The fork method creates a new sequence from an existing one. The append method appends a new token to the sequence. Finally, the free method deletes the sequence. For instance, in parallel sampling, vLLM creates multiple output sequences from the single input sequence using the fork method. It then adds new tokens to these sequences in every iteration with append, and deletes sequences that meet a stopping condition using free. The same strategy is also applied in beam search and prefix sharing by vLLM. We believe future decoding algorithms can also be supported by combining these methods.

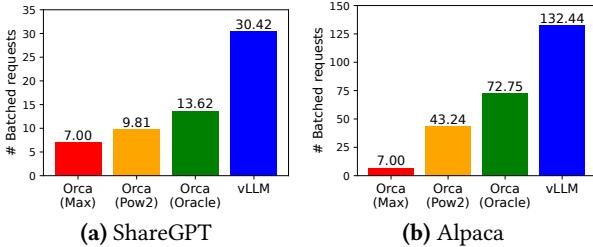
## 6 Evaluation

In this section, we evaluate the performance of vLLM under a variety of workloads.

利用读缓存，  
写不缓存的特性



**Figure 12.** Single sequence generation with OPT models on the ShareGPT and Alpaca dataset



**Figure 13.** Average number of batched requests when serving OPT-13B for the ShareGPT (2 reqs/s) and Alpaca (30 reqs/s) traces.

## 6.1 Experimental Setup

**Model and server configurations.** We use OPT [62] models with 13B, 66B, and 175B parameters and LLaMA [52] with 13B parameters for our evaluation. 13B and 66B are popular sizes for LLMs as shown in an LLM leaderboard [38], while 175B is the size of the famous GPT-3 [5] model. For all of our experiments, we use A2 instances with NVIDIA A100 GPUs on Google Cloud Platform. The detailed model sizes and server configurations are shown in Table 1.

**Workloads.** We synthesize workloads based on ShareGPT [51] and Alpaca [50] datasets, which contain input and output texts of real LLM services. The ShareGPT dataset is a collection of user-shared conversations with ChatGPT [35]. The Alpaca dataset is an instruction dataset generated by GPT-3.5 with self-instruct [57]. We tokenize the datasets and use their input and output lengths to synthesize client requests. As shown in Fig. 11, the ShareGPT dataset has 8.4× longer input prompts and 5.8× longer outputs on average than the Alpaca dataset, with higher variance. Since these datasets do not include timestamps, we generate request arrival times using Poisson distribution with different request rates.

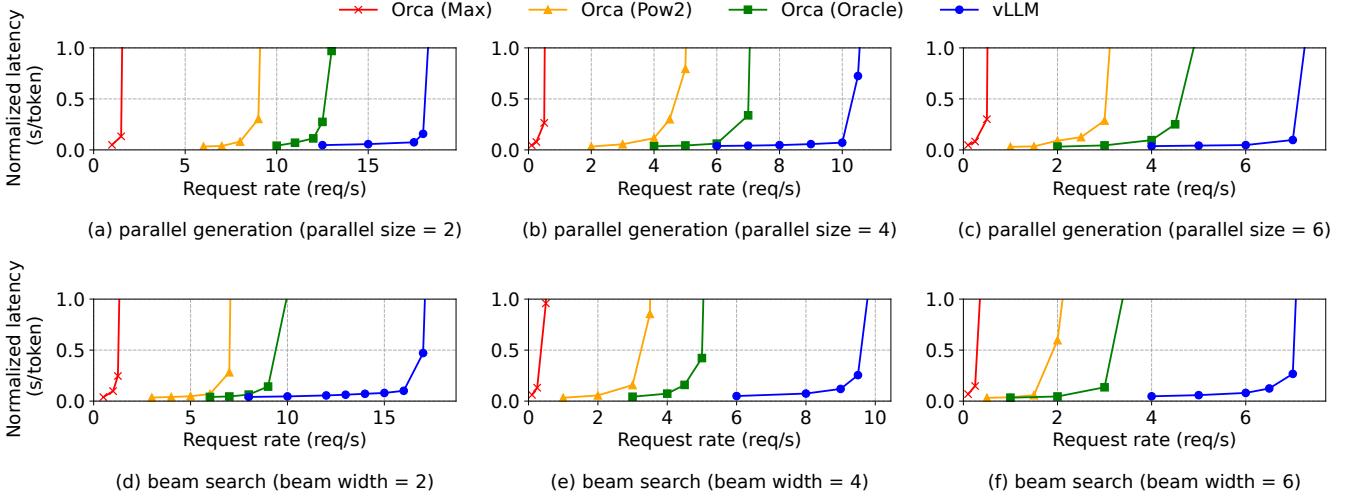
**Baseline 1: FasterTransformer.** FasterTransformer [31] is a distributed inference engine highly optimized for latency.

As FasterTransformer does not have its own scheduler, we implement a custom scheduler with a dynamic batching mechanism similar to the existing serving systems such as Triton [30]. Specifically, we set a maximum batch size  $B$  as large as possible for each experiment, according to the GPU memory capacity. The scheduler takes up to  $B$  number of earliest arrived requests and sends the batch to FasterTransformer for processing.

**Baseline 2: Orca.** Orca [60] is a state-of-the-art LLM serving system optimized for throughput. Since Orca is not publicly available for use, we implement our own version of Orca. We assume Orca uses the buddy allocation algorithm to determine the memory address to store KV cache. We implement three versions of Orca based on how much it over-reserves the space for request outputs:

- **Orca (Oracle).** We assume the system has the knowledge of the lengths of the outputs that will be actually generated for the requests. This shows the upper-bound performance of Orca, which is infeasible to achieve in practice.
- **Orca (Pow2).** We assume the system over-reserves the space for outputs by at most 2x. For example, if the true output length is 25, it reserves 32 positions for outputs.
- **Orca (Max).** We assume the system always reserves the space up to the maximum sequence length of the model, i.e., 2048 tokens.

**Key metrics.** We focus on serving throughput. Specifically, using the workloads with different request rates, we measure *normalized latency* of the systems, the mean of every request's end-to-end latency divided by its output length, as in Orca [60]. A high-throughput serving system should retain low normalized latency against high request rates. For most experiments, we evaluate the systems with 1-hour traces. As an exception, we use 15-minute traces for the OPT-175B model due to the cost limit.



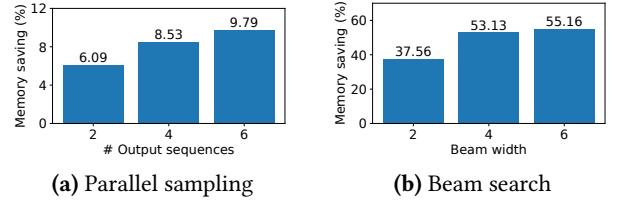
**Figure 14.** Parallel generation and beam search with OPT-13B on the Alpaca dataset.

## 6.2 Basic Sampling

We evaluate the performance of vLLM with basic sampling (**one sample per request**) on three models and two datasets. The first row of Fig. 12 shows the results on the ShareGPT dataset. The curves illustrate that as the request rate increases, the latency initially increases at a gradual pace but then suddenly explodes. This can be attributed to the fact that when the request rate surpasses the capacity of the serving system, the queue length continues to grow infinitely and so does the latency of the requests.

On the ShareGPT dataset, vLLM can sustain  $1.7\times$ - $2.7\times$  higher request rates compared to Orca (Oracle) and  $2.7\times$ - $8\times$  compared to Orca (Max), while maintaining similar latencies. This is because vLLM’s PagedAttention can efficiently manage the memory usage and thus enable batching more requests than Orca. For example, as shown in Fig. 13a, for OPT-13B vLLM processes  $2.2\times$  more requests at the same time than Orca (Oracle) and  $4.3\times$  more requests than Orca (Max). Compared to FasterTransformer, vLLM can sustain up to  $22\times$  higher request rates, as FasterTransformer does not utilize a fine-grained scheduling mechanism and inefficiently manages the memory like Orca (Max).

The second row of Fig. 12 and Fig. 13b shows the results on the Alpaca dataset, which follows a similar trend to the ShareGPT dataset. One exception is Fig. 12 (f), where vLLM’s advantage over Orca (Oracle) and Orca (Pow2) is less pronounced. This is because the model and server configuration for OPT-175B (Table 1) allows for large GPU memory space available to store KV cache, while the Alpaca dataset has short sequences. In this setup, Orca (Oracle) and Orca (Pow2) can also batch a large number of requests despite the inefficiencies in their memory management. As a result, the performance of the systems becomes compute-bound rather than memory-bound.



**Figure 15.** Average amount of memory saving from sharing KV blocks, when serving OPT-13B for the Alpaca trace.

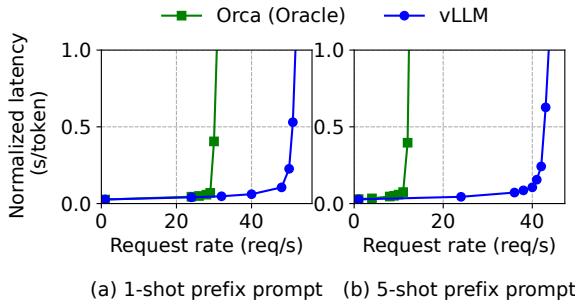
## 6.3 Parallel Sampling and Beam Search

We evaluate the effectiveness of memory sharing in Page-dAttention with two popular sampling methods: parallel sampling and beam search. In parallel sampling, all parallel sequences in a request can share the KV cache for the prompt. As shown in the first row of Fig. 14, with a larger number of sequences to sample, vLLM brings more improvement over the Orca baselines. Similarly, the second row of Fig. 14 shows the results for beam search with different beam widths. Since beam search allows for more sharing, vLLM demonstrates even greater performance benefits. The improvement of vLLM over Orca (Oracle) on OPT-13B and the Alpaca dataset goes from  $1.3\times$  in basic sampling to  $2.3\times$  in beam search with a width of 6.

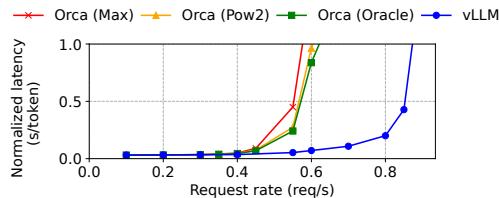
Fig. 15 plots the amount of memory saving, computed by the number of blocks we saved by sharing divided by the number of total blocks without sharing. We show 6.1% - 9.8% memory saving on parallel sampling and 37.6% - 55.2% on beam search. In the same experiments with the ShareGPT dataset, we saw 16.2% - 30.5% memory saving on parallel sampling and 44.3% - 66.3% on beam search.

## 6.4 Shared prefix

We explore the effectiveness of vLLM for the case a prefix is shared among different input prompts, as illustrated in



**Figure 16.** Translation workload where the input prompts share a common prefix. The prefix includes (a) 1 example with 80 tokens or (b) 5 examples with 341 tokens.



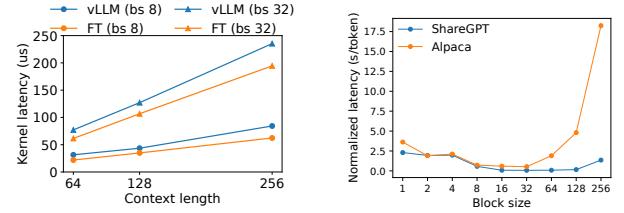
**Figure 17.** Performance on chatbot workload.

Fig. 10. For the model, we use LLaMA-13B [52], which is multilingual. For the workload, we use the WMT16 [4] English-to-German translation dataset and synthesize two prefixes that include an instruction and a few translation examples. The first prefix includes a single example (i.e., one-shot) while the other prefix includes 5 examples (i.e., few-shot). As shown in Fig. 16 (a), vLLM achieves 1.67 $\times$  higher throughput than Orca (Oracle) when the one-shot prefix is shared. Furthermore, when more examples are shared (Fig. 16 (b)), vLLM achieves 3.58 $\times$  higher throughput than Orca (Oracle).

## 6.5 Chatbot

A chatbot [8, 19, 35] is one of the most important applications of LLMs. To implement a chatbot, we let the model generate a response by concatenating the chatting history and the last user query into a prompt. We synthesize the chatting history and user query using the ShareGPT dataset. Due to the limited context length of the OPT-13B model, we cut the prompt to the last 1024 tokens and let the model generate at most 1024 tokens. We do not store the KV cache between different conversation rounds as doing this would occupy the space for other requests between the conversation rounds.

Fig. 17 shows that vLLM can sustain 2 $\times$  higher request rates compared to the three Orca baselines. Since the ShareGPT dataset contains many long conversations, the input prompts for most requests have 1024 tokens. Due to the buddy allocation algorithm, the Orca baselines reserve the space for 1024 tokens for the request outputs, regardless of how they predict the output lengths. For this reason, the three Orca baselines behave similarly. In contrast, vLLM can effectively



(a) Latency of attention kernels. (b) End-to-end latency with different block sizes.

**Figure 18.** Ablation experiments.

handle the long prompts, as PagedAttention resolves the problem of memory fragmentation and reservation.

## 7 Ablation Studies

In this section, we study various aspects of vLLM and evaluate the design choices we make with ablation experiments.

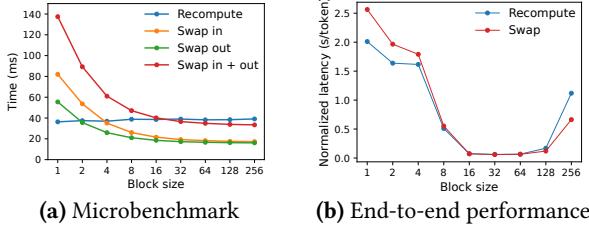
### 7.1 Kernel Microbenchmark

The dynamic block mapping in PagedAttention affects the performance of the GPU operations involving the stored KV cache, i.e., block read/writes and attention. Compared to the existing systems, our GPU kernels (§5) involve extra overheads of accessing the block table, executing extra branches, and handling variable sequence lengths. As shown in Fig. 18a, this leads to 20–26% higher attention kernel latency, compared to the highly-optimized FasterTransformer implementation. We believe the overhead is small as it only affects the attention operator but not the other operators in the model, such as Linear. Despite the overhead, PagedAttention makes vLLM significantly outperform FasterTransformer in end-to-end performance (§6).

### 7.2 Impact of Block Size

The choice of block size can have a substantial impact on the performance of vLLM. If the block size is too small, vLLM may not fully utilize the GPU’s parallelism for reading and processing KV cache. If the block size is too large, internal fragmentation increases and the probability of sharing decreases.

In Fig. 18b, we evaluate the performance of vLLM with different block sizes, using the ShareGPT and Alpaca traces with basic sampling under fixed request rates. In the ShareGPT trace, block sizes from 16 to 128 lead to the best performance. In the Alpaca trace, while the block size 16 and 32 work well, larger block sizes significantly degrade the performance since the sequences become shorter than the block sizes. In practice, we find that the block size 16 is large enough to efficiently utilize the GPU and small enough to avoid significant internal fragmentation in most workloads. Accordingly, vLLM sets its default block size as 16.



**Figure 19.** (a) Overhead of recomputation and swapping for different block sizes. (b) Performance when serving OPT-13B with the ShareGPT traces at the same request rate.

### 7.3 Comparing Recomputation and Swapping

vLLM supports both recomputation and swapping as its recovery mechanisms. To understand the tradeoffs between the two methods, we evaluate their end-to-end performance and microbenchmark their overheads, as presented in Fig. 19. Our results reveal that **swapping incurs excessive overhead with small block sizes**. This is because small block sizes often result in numerous small data transfers between CPU and GPU, which limits the effective PCIe bandwidth. **In contrast, the overhead of recomputation remains constant across different block sizes, as recomputation does not utilize the KV blocks.** Thus, **recomputation is more efficient when the block size is small**, while swapping is more efficient when the block size is large, though recomputation overhead is never higher than 20% of swapping’s latency. For medium block sizes from 16 to 64, the two methods exhibit comparable end-to-end performance.

## 8 Discussion

**Applying the virtual memory and paging technique to other GPU workloads.** The idea of virtual memory and paging is effective for managing the KV cache in LLM serving because the workload requires dynamic memory allocation (since the output length is not known a priori) and its performance is bound by the GPU memory capacity. However, this does not generally hold for every GPU workload. For example, **in DNN training, the tensor shapes are typically static**, and thus memory allocation can be optimized ahead of time. For another example, **in serving DNNs that are not LLMs**, an increase in memory efficiency may not result in any performance improvement since the performance is primarily compute-bound. **In such scenarios, introducing the vLLM’s techniques may rather degrade the performance due to the extra overhead of memory indirection and non-contiguous block memory.** However, we would be excited to see vLLM’s techniques being applied to other workloads with similar properties to LLM serving.

**LLM-specific optimizations in applying virtual memory and paging.** vLLM re-interprets and augments the idea of virtual memory and paging by leveraging the application-specific semantics. One example is vLLM’s all-or-nothing

swap-out policy, which exploits the fact that processing a request requires all of its corresponding token states to be stored in GPU memory. Another example is the recomputation method to recover the evicted blocks, which is not feasible in OS. Besides, vLLM mitigates the overhead of memory indirection in paging by fusing the GPU kernels for memory access operations with those for other operations such as attention. 缓和

## 9 Related Work

**General model serving systems.** Model serving has been an active area of research in recent years, with numerous systems proposed to tackle diverse aspects of deep learning model deployment. Clipper [11], TensorFlow Serving [33], Nexus [45], InferLine [10], and Clockwork [20] are some earlier general model serving systems. They study batching, caching, placement, and scheduling for serving single or multiple models. More recently, DVABatch [12] introduces multi-entry multi-exit batching. REEF [21] and Shepherd [61] propose preemption for serving. AlpaServe [28] utilizes model parallelism for statistical multiplexing. However, these general systems fail to take into account the auto-regressive property and token state of LLM inference, resulting in missed opportunities for optimization.

**Specialized serving systems for transformers.** Due to the significance of the transformer architecture, numerous specialized serving systems for it have been developed. These systems utilize GPU kernel optimizations [1, 29, 31, 56], advanced batching mechanisms [14, 60], model parallelism [1, 41, 60], and parameter sharing [64] for efficient serving. Among them, **Orca** [60] is most relevant to our approach.

**Comparison to Orca.** The iteration-level scheduling in Orca [60] and PagedAttention in vLLM are complementary techniques: While both systems aim to increase the GPU utilization and hence the throughput of LLM serving, Orca achieves it by scheduling and interleaving the requests so that more requests can be processed in parallel, while vLLM is doing so by increasing memory utilization so that the working sets of more requests fit into memory. By reducing memory fragmentation and enabling sharing, **vLLM runs more requests in a batch in parallel and achieves a 2-4× speedup compared to Orca**. Indeed, the fine-grained scheduling and interleaving of the requests like in Orca makes memory management more challenging, making the techniques proposed in vLLM even more crucial.

**Memory optimizations.** The widening gap between the compute capability and memory capacity of accelerators has caused memory to become a bottleneck for both training and inference. **Swapping** [23, 42, 55], **recomputation** [7, 24] and their combination [40] have been utilized to reduce the peak memory of training. Notably, FlexGen [46] studies how to **swap weights and token states for LLM inference with**

limited GPU memory, but it does not target the online serving settings. **OLLA** [48] optimizes the lifetime and location of tensors to reduce fragmentation, but it does not do fine-grained block-level management or online serving. **FlashAttention** [13] applies tiling and kernel optimizations to reduce the peak memory of attention computation and reduce I/O costs. This paper introduces a new idea of block-level memory management in the context of online serving.

## 10 Conclusion

This paper proposes PagedAttention, a new attention algorithm that allows attention keys and values to be stored in non-contiguous paged memory, and presents vLLM, a high-throughput LLM serving system with efficient memory management enabled by PagedAttention. Inspired by operating systems, we demonstrate how established techniques, such as virtual memory and copy-on-write, can be adapted to efficiently manage KV cache and handle various decoding algorithms in LLM serving. Our experiments show that vLLM achieves 2-4 $\times$  throughput improvements over the state-of-the-art systems.

## Acknowledgement

We would like to thank Xiaoxuan Liu, Zhifeng Chen, Yanping Huang, anonymous SOSP reviewers, and our shepherd, Lidong Zhou, for their insightful feedback. This research is partly supported by gifts from Andreessen Horowitz, Anyscale, Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, Uber, and VMware.

## References

- [1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, et al. 2022. DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. *arXiv preprint arXiv:2207.00032* (2022).
- [2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).
- [3] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. 2000. A neural probabilistic language model. *Advances in neural information processing systems* 13 (2000).
- [4] Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Aurelie Neveol, Mariana Neves, Martin Popel, Matt Post, Raphael Rubino, Carolina Scarton, Lucia Specia, Marco Turchi, Karin Verspoor, and Marcos Zampieri. 2016. Findings of the 2016 Conference on Machine Translation. In *Proceedings of the First Conference on Machine Translation*. Association for Computational Linguistics, Berlin, Germany, 131–198. <http://www.aclweb.org/anthology/W/W16/W16-2301>
- [5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).
- [8] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%\* ChatGPT Quality. <https://lmsys.org/blog/2023-03-30-vicuna/>
- [9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).
- [10] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 477–491.
- [11] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 613–627.
- [12] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. 2022. DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 183–198.
- [13] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.
- [14] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: an efficient GPU serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 389–402.
- [15] FastAPI. 2023. FastAPI. <https://github.com/tiangolo/fastapi>.
- [16] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.
- [17] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W Mahoney, and Kurt Keutzer. 2021. Ai and memory wall. *RiseLab Medium Post* 1 (2021), 6.
- [18] Github. 2022. <https://github.com/features/copilot>
- [19] Google. 2023. <https://bard.google.com/>
- [20] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving {DNNs} like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 443–462.
- [21] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent {GPU-accelerated} {DNN} Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 539–558.
- [22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
- [23] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1341–1355.
- [24] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the memory wall with optimal tensor rematerialization.

- Proceedings of Machine Learning and Systems* 2 (2020), 497–511.
- [25] Tom Kilburn, David BG Edwards, Michael J Lanigan, and Frank H Sumner. 1962. One-level storage system. *IRE Transactions on Electronic Computers* 2 (1962), 223–235.
- [26] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691* (2021).
- [27] Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190* (2021).
- [28] Zhuohan Li, Lianmin Zheng, Yimin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. *arXiv preprint arXiv:2302.11665* (2023).
- [29] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 881–897.
- [30] NVIDIA. [n. d.]. Triton Inference Server. <https://developer.nvidia.com/nvidia-triton-inference-server>.
- [31] NVIDIA. 2023. FasterTransformer. <https://github.com/NVIDIA/FasterTransformer>.
- [32] NVIDIA. 2023. NCCL: The NVIDIA Collective Communication Library. <https://developer.nvidia.com/nccl>.
- [33] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Tao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyeke. 2017. Tensorflow-serving: Flexible, high-performance m1 serving. *arXiv preprint arXiv:1712.06139* (2017).
- [34] OpenAI. 2020. <https://openai.com/blog/openai-api>
- [35] OpenAI. 2022. <https://openai.com/blog/chatgpt>
- [36] OpenAI. 2023. <https://openai.com/blog/custom-instructions-for-chatgpt>
- [37] OpenAI. 2023. GPT-4 Technical Report. *arXiv:2303.08774* [cs.CL]
- [38] LMSYS ORG. 2023. Chatbot Arena Leaderboard Week 8: Introducing MT-Bench and Vicuna-33B. <https://lmsys.org/blog/2023-06-22-leaderboard/>.
- [39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
- [40] Shishir G Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph Gonzalez. 2022. POET: Training Neural Networks on Tiny Devices with Integrated Rematerialization and Paging. In *International Conference on Machine Learning*. PMLR, 17573–17583.
- [41] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scaling Transformer Inference. *arXiv preprint arXiv:2211.05102* (2022).
- [42] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training.. In *USENIX Annual Technical Conference*. 551–564.
- [43] Reuters. 2023. <https://www.reuters.com/technology/tech-giants-ai-like-bing-bard-poses-billion-dollar-search-problem-2023-02-22/>
- [44] Amazon Web Services. 2023. <https://aws.amazon.com/bedrock/>
- [45] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 322–337.
- [46] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. 2023. High-throughput Generative Inference of Large Language Models with a Single GPU. *arXiv preprint arXiv:2303.06865* (2023).
- [47] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).
- [48] Benoit Steiner, Mostafa Elhoushi, Jacob Kahn, and James Hegarty. 2022. OLLA: Optimizing the Lifetime and Location of Arrays to Reduce the Memory Usage of Neural Networks. (2022). <https://doi.org/10.48550/arXiv.2210.12924>
- [49] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).
- [50] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. [https://github.com/tatsu-lab/stanford\\_alpaca](https://github.com/tatsu-lab/stanford_alpaca).
- [51] ShareGPT Team. 2023. <https://sharegpt.com/>
- [52] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
- [53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
- [54] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. 2022. Pacman: An Efficient Compaction Approach for {Log-Structured} {Key-Value} Store on Persistent Memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 773–788.
- [55] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 41–53.
- [56] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. 2021. LightSeq: A High Performance Inference Library for Transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*. 113–120.
- [57] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-Instruct: Aligning Language Model with Self Generated Instructions. *arXiv preprint arXiv:2212.10560* (2022).
- [58] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierrick Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 38–45.
- [59] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).
- [60] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for {Transformer-Based} Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.
- [61] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 787–808. <https://www.usenix.org/conference/nsdi23/presentation/zhang-hong>

- [62] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuhui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).
- [63] Lianmin Zheng, Zhuhan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.
- [64] Zhe Zhou, Xuechao Wei, Jiebing Zhang, and Guangyu Sun. 2022. PetS: A Unified Framework for Parameter-Efficient Transformers Serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 489–504.

# Reducing shared memory footprint to leverage high throughput on Tensor Cores and its flexible API extension library

Hiroyuki Ootomo  
 Tokyo Institute of Technology  
 Tokyo, Japan  
 ootomo.h@rio.gsic.titech.ac.jp

Rio Yokota  
 Tokyo Institute of Technology  
 Tokyo, Japan

## Abstract

Matrix-matrix multiplication is used for various linear algebra algorithms such as matrix decomposition and tensor contraction. NVIDIA Tensor Core is a mixed-precision matrix-matrix multiplication and addition computing unit, where the theoretical peak performance is more than 300 TFlop/s on NVIDIA A100 GPU. NVIDIA provides WMMA API for using Tensor Cores in custom kernel functions. The most common way to use Tensor Core is to supply the input matrices from shared memory, which has higher bandwidth than global memory. However, the Bytes-per-Flops (B/F) ratio of the shared memory and Tensor Cores is small since the performance of Tensor Cores is high. Thus, it is important to reduce the shared memory footprint for efficient Tensor Cores usage. In this paper, we analyze the simple matrix-matrix multiplication on Tensor Cores by the roofline model and figure out that the bandwidth of shared memory might be a limitation of the performance when using WMMA API. To alleviate this issue, we provide a WMMA API extension library to boost the throughput of the computation, which has two components. The first one allows for manipulating the array of registers input to Tensor Cores flexibly. We evaluate the performance improvement of this library. The outcome of our evaluation shows that our library reduces the shared memory footprint and speeds up the computation using Tensor Cores. The second one is an API for the SGEMM emulation on Tensor Cores without additional shared memory usage. We have demonstrated that the single-precision emulating batch SGEMM implementation on Tensor Cores using this library achieves 54.2 TFlop/s on A100 GPU, which outperforms the theoretical peak performance of FP32 SIMT

Cores while achieving the same level of accuracy as cuBLAS. The achieved throughput can not be achieved without reducing the shared memory footprint done by our library with the same amount of register usage.

**CCS Concepts:** • Software and its engineering → Software libraries and repositories.

**Keywords:** Tensor Cores, WMMA API, GPU

## ACM Reference Format:

Hiroyuki Ootomo and Rio Yokota. 2023. Reducing shared memory footprint to leverage high throughput on Tensor Cores and its flexible API extension library. In *International Conference on High Performance Computing in Asia-Pacific Region (HPC ASIA 2023), February 27–March 2, 2023, Singapore, Singapore*. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3578178.3578238>

## 1 Introduction

NVIDIA Tensor Core is a mixed-precision matrix multiplication and addition computing unit with up to 312 TFlop/s on NVIDIA A100 GPU [2]. From the demand for high-throughput matrix multiplication from deep learning, several computing units specialized for matrix multiplication are developed, such as Google TPU [7], AMD Matrix Core, Intel Ponte Vecchio, and Preferred Networks MN-Core [9]. Tensor Core computes the multiplication of two matrices where the data type is low-precision in high throughput and high-precision. Although Tensor Core is developed for deep learning, especially fully-connected layer and convolution layer computations, it is applied to other fields of computations and fundamental linear algebra algorithms leveraging the low- and mixed-precision feature [3–5, 8, 10, 11]. NVIDIA provides highly optimized libraries for using Tensor Cores which can be called from a host, such as cuBLAS and cuDNN. We can leverage the high throughput of Tensor Core using these libraries without special knowledge of it. Furthermore, NVIDIA also provides an API for use inside a CUDA kernel function called WMMA (Warp Matrix Multiply Accumulate) API. This API provides basic functionalities such as loading matrix data from memory, multiplication and addition on Tensor Core, and storing the resulting matrix data in memory. Using this API, we load matrix data from the device memory or shared memory to an array of registers called “fragment” to input

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

HPC ASIA 2023, February 27–March 2, 2023, Singapore, Singapore

© 2023 Association for Computing Machinery.

ACM ISBN 978-1-4503-9805-3/23/02...\$15.00

<https://doi.org/10.1145/3578178.3578238>

Tensor Cores. On the other hand, there are some matrices where each element can be computed on the fly, for instance, the Householder matrix and Given’s rotation matrix. Even for these matrices, we have to store them in memory and load them since the API is too simple and lacks flexibility, and this can degrade the throughput. Thus, for instance, Dakkak *et al.* [3] use Tensor Cores for reduction and scan operation by generating a fragment of an upper triangular matrix and a lower triangular matrix without generating the matrices on the shared memory. Li *et al.* [8] use Tensor Cores in FFT operations by generating fragments directly. However, since NVIDIA does not provide information on fragment mapping, we need to analyze the structure of the fragment by ourselves to generate the fragment in these ways. For another example, the single-precision matrix-matrix multiplication emulation method on Tensor Cores [11] accesses the shared memory more than necessary if we only use the WMMA API. Therefore, the throughput of the emulation method can degrade if we only use the API.

In this paper, we first show that it is important to reduce the shared memory footprint to leverage the high Tensor Cores performance. We analyze a matrix-matrix multiplication on Tensor Cores using the roofline model [12]. As a result, it is difficult to leverage the high Tensor Cores performance without sufficient register blocking or reducing the shared memory footprint. However, the number of registers is limited. To reduce the shared memory footprint, we implement a WMMA API extension library, which flexibly manipulates the input register array of Tensor Cores by analyzing the memory and register array mappings. This library can generate an arbitrary input register array without an extra shared memory footprint. Furthermore, we provide an API for single-precision matrix-matrix multiplication emulation on Tensor Cores, which has the same interface as WMMA API. Our goals of this work are 1) to reveal that the shared memory bandwidth can degrade the utilization efficiency of Tensor Cores in some cases and 2) to provide a library to reduce such degradation by manipulating the fragment flexibly for reducing the shared memory footprint.

Our contributions are as follows:

- We show that the shared memory bandwidth might limit the matrix-matrix multiplication performance on Tensor Cores by roofline model analysis. Furthermore, we find it important to reduce the shared memory footprint on NVIDIA A100 compared to V100 since the Bytes-per-Flops (B/F) ratio of the Tensor Core performance and shared memory bandwidth on NVIDIA A100 is smaller than V100.
- We implement a general WMMA API extension library to reduce the shared memory footprint. By using this library, we can manipulate the fragment elements flexibly. And as a secondary effect, we can reduce the shared memory usage in some cases since some of the

	V100 (SXM2)	V100S (PCIe)	A100 (SXM4/PCIe)
SMs	80	108	
Clock [MHz]	1,380	1,597	1,410
<b>Device memory</b>			
Size [GB]	32/16	32	40
Bandwidth [GB/s]	900	1,134	1,555
<b>Shared memory</b>			
Size [KB/SM]	~96	~164	
Bandwidth [GB/s]	14,131	16,353	19,491
<b>Performance</b>			
FP32 [TFlop/s]	15.7	16.4	19.5
FP16 [TFlop/s]	31.4	32.8	39.0
FP16-TC [TFlop/s]	112	125	312
TF32-TC [TFlop/s]	-	-	156

Table 1. Specifications of NVIDIA GPUs A100 and V100.

temporary shared memory areas for generating matrices that are loaded as fragments become unnecessary. We investigate the availability of this library and find the condition to speed up the fragment generation. The library is available on GitHub<sup>1</sup>.

- We figure out that by the inflexibility of WMMA API, shared memory bandwidth bounds the theoretical peak performance of single-precision matrix-matrix multiplication emulation on Tensor Cores. By using our extension library, we improve its theoretical peak performance. Furthermore, we provide functionality for that which has the same interface as WMMA API. To demonstrate the usability of the functionality, we implement batched matrix-matrix multiplication using the functionality. We show that our implementation outperforms the FP32 theoretical peak performance on NVIDIA A100 while the accuracy is the same level as cuBLAS SGEMM.

## 2 Background

### 2.1 Shared memory

**2.1.1 The bandwidth of shared memory.** The shared memory is a high bandwidth, low latency, and small size compared to the device memory. This memory is located on each Streaming Multiprocessor (SM) and shared by all threads in a thread block. The shared memory is divided into the same size memory modules called banks. In CUDA, a cluster of threads consisting of 32 threads is called a warp, and when multiple threads in a warp access the same bank and different addresses, it is called bank conflict. Since bank conflict degrades read/write performance, there are known

<sup>1</sup>[https://github.com/wmmae/wmma\\_extension](https://github.com/wmmae/wmma_extension)

workarounds, such as shifting the boundaries of shared memory. We show the specifications of NVIDIA Tesla V100 and A100 in Table 1. The shared memory bandwidth is calculated assuming it is accessed without bank conflict in all SMs in one clock. The shared memory has 12 ~ 15 times faster bandwidth than device memory.

### 2.1.2 The advantage of fewer shared memory usage.

The shared memory size that one thread block uses is one of the determining factors of occupancy, which is the max thread block size that one SM executes simultaneously. Fewer shared memory usage means higher occupancy, which effectively hides instruction latency. Furthermore, reducing shared memory usage can improve the L1 cache hit rate since shared memory and L1 cache resides in the same part of the chip.

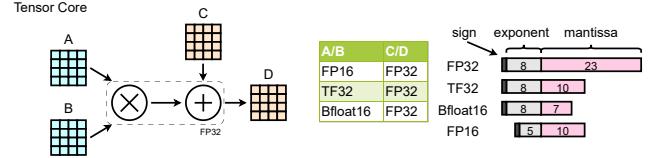
## 2.2 Blocking for matrix-matrix multiplication

The number of operations of matrix-matrix multiplication  $C \leftarrow A \cdot B$  for  $A \in \mathbb{R}^{m \times k}, B \in \mathbb{R}^{k \times n}$  is  $2mnk$ . On the other hand, the sum of the number of elements in  $A$  and  $B$  is  $(m + n) \times k$ . It follows  $2mnk > (m + n) \times k$  in general ( $m \geq 2, n \geq 2, k \geq 1$ ), which means that the number of operations is larger than the number of data. Thus, data can be reused during the computation. When computing the matrix-matrix multiplication on device memory, we copy the sub-matrices of each input matrix from device memory to shared memory. Then compute the matrix-matrix multiplication of these sub-matrices on shared memory to reduce the device memory footprint using data reusability. This method of reducing the low-bandwidth memory footprint by utilizing the memory hierarchy is called “blocking”. The registers are also used for blocking the shared memory. In this paper, we denote the blocking size  $(m_b, n_b, k_b)$  as the size of blocking size for sub-matrices matrix-matrix multiplication  $A_b \cdot B_b$  where the sizes of matrix  $A_b$  and  $B_b$  are  $m_b \times k_b$  and  $k_b \times n_b$  respectively.

## 2.3 Tensor Cores

Tensor Cores are specialized computing units for mixed-precision matrix-matrix multiplication and addition, with higher computing performance than FP16 and FP32 computing units shown in Table 1. We show the supported input and output data types of Tensor Core in Figure 1. We can use the TF32 (Tensor Float) data type, 8 bits of exponent and 10 bits of mantissa, and Bfloat16, 8 bits of exponent and 7 bits of mantissa, as inputs to Tensor Cores in Ampere architectures. While TF32 has 19 bits in total, it occupies a 32-bit register and memory. Thus, it can not be used for data compression.

**2.3.1 Programming interface.** To use Tensor Cores in custom functions, NVIDIA provides WMMA API for C++ and Parallel Thread Execution (PTX). When computing matrix-matrix multiplication and addition  $D \leftarrow A \cdot B + C$  on Tensor Cores using WMMA API for C++, first, we copy the input matrices  $A, B$  and,  $C$  from memory to an array of registers



**Figure 1.** The input and output types of Tensor Cores on NVIDIA A100.

called “fragment”. Then, we compute Matrix-Multiplication-and-Add (MMA) on the Tensor Cores and obtain the resulting  $D$  fragment. The 32 threads in a warp cooperate to perform MMA operations on Tensor Cores. Finally, we store the  $D$  fragment in memory. The WMMA API provides the fragment and functions for these operations. The fragment is a C language structure that has an array of registers  $x[num\_elements]$  as a member. We show the pseudocode of simple matrix-matrix multiplication using WMMA API in Code 1.

```

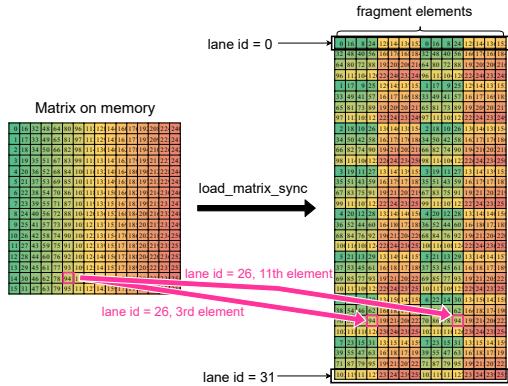
1 __device__
2 void matmul( float* mem_c, half* mem_a, half* mem_b
    ) {
3     using namespace nvcuda::wmma;
4     fragment<matrix_a, 16, 16, 16, half, col_major>
        frag_a;
5     fragment<matrix_b, 16, 16, 16, half, col_major>
        frag_b;
6     fragment<accumulator, 16, 16, 16, float> frag_c;
7     // Initialize an accumulator fragment
8     fill_fragment(frag_c, 0.f);
9     // Load matrices to fragments
10    load_matrix_sync(frag_a, mem_a, ...);
11    load_matrix_sync(frag_b, mem_b, ...);
12    // Compute matrix-matrix multiplication
13    // and accumulation on Tensor Cores
14    mma_sync(frag_c, frag_a, frag_b, frag_c);
15    // Store result to memory
16    store_matrix_sync(mem_c, frag_c, ...);
17 }
```

**Code 1.** A simple matrix-matrix multiplication on Tensor Cores using WMMA API.

Although the `load_matrix_sync` function in WMMA API can generate a fragment from the device and shared memory, we consider that the shared memory is used in most cases for the following reasons:

- The shared memory is used for memory blocking in matrix-matrix multiplication.
- The `load_matrix_sync` function has a 128-bit alignment restriction and leading dimension size restriction. It is difficult to satisfy the restriction on device memory.

The fragment is regarded as a register blocking. WMMA API specifies the blocking size of one fragment. For instance, in the case of FP16-Tensor Core, the blocking size  $(m_b, n_b, k_b)$



**Figure 2.** An example of memory-fragment mapping. The lane id is a thread number in a warp which is calculated by  $(\text{threadIdx.x} \& 0x1f)$ .

is one of the (16, 16, 16), (32, 8, 16) or (8, 32, 16). We can use the array of fragments to increase the blocking size.

**2.3.2 Mapping between memory and fragment.** Each matrix element in memory is stored as an element of a fragment of some thread. Although the mapping between memory and fragment elements is not public, we can investigate it [3, 6, 8]. This mapping depends on the type, memory layout, etc, of the matrix. We use Code 2 to investigate the mapping and show an example of the mapping in Figure 2.

```

1 template <class Use, class Layout, class T>
2 __global__ void investigate_mapping() {
3     __shared__ T smem[];
4     // initialize smem
5     for (i = 0; i < 16 * 16; i++) smem[i] = i;
6     fragment<Use, 16, 16, 16, T, Layout> frag;
7     load_matrix_sync(frag, smem, ...); // WMMA API
8     for (i = 0; i < 32; i++) {
9         if (threadIdx.x == i){
10             for (j = 0; j < frag.num_elements; j++) {
11                 // Print the mapping
12                 printf("%d, ", (int)frag.x[j]);
13                 printf("\n");
14             __syncwarp();
15         }
16     }

```

**Code 2.** A kernel function to investigate the memory-fragment mappings.

**2.3.3 WMMA API for PTX.** The WMMA API for PTX provides two types of instructions: 1) wmma instructions and 2) mma instruction. The WMMA API for C++ functions calls wmma instructions using inline assembly. The wmma instructions include functionality for loading and storing fragments and MMA operation. On the other hand, mma instruction only includes MMA operation. Thus, when using mma instruction, we must manually load fragments from memory. The mapping is available on CUDA developer

documentation. There is a difference between the wmma instructions and the mma instruction regarding register usage. When using wmma instructions, one element in a matrix is kept by two elements in a fragment in 32 threads in a warp. On the other hand, when using mma instruction, one element in a matrix is kept by only one element in a fragment in 32 threads in a warp without duplication. Thus, the mma instruction computes MMA operation using fewer registers than the wmma instructions.

### 3 The balance of Tensor Cores performance and shared memory bandwidth

Although the shared memory bandwidth is higher than device memory, the computing performance of the Tensor Cores is high, and its Bytes-per-Flops (B/F) ratio is calculated to be  $0.06 \sim 0.12$  from Table 1. This value is similar to the ratio between the FP32 computing unit and device memory ( $0.06 \sim 0.10$ ). In the case of the FP32 computing unit and device memory, the memory blocking using shared memory reduces global memory access and alleviates the problem of this small B/F ratio. Similarly, in the case of shared memory and Tensor Cores, it is important to reduce shared memory accesses to take advantage of high computational performance.

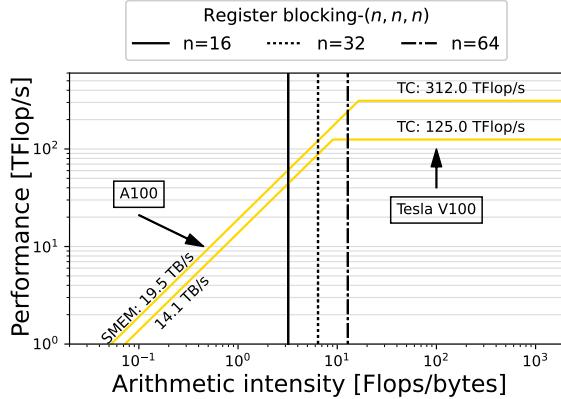
Now, we analyze a matrix-matrix multiplication on Tensor Cores using the roofline model. The input matrices A and B are FP16, C and D are FP32 stored in the shared memory. We load the sub-matrices of each input matrix as fragments  $A_{\text{reg}}$  and  $B_{\text{reg}}$  for register blocking. The register blocking size is  $(n, n, n)$ . We show the roofline model of computing  $D_{\text{reg}} \leftarrow A_{\text{reg}} \cdot B_{\text{reg}} + C_{\text{reg}}$  in Figure 1. The Arithmetic Intensity (AI) is calculated as follows:

$$\text{AI} = \frac{2n^3}{(n^2 + n^2)\text{sizeof(FP16)} + (n^2 + n^2)\text{sizeof(FP32)}} = \frac{n}{5}. \quad (1)$$

As the size of register blocking size increases, we can utilize the performance of Tensor Cores more. However, the number of registers is finite, and the registers spill to local memory when using more than 256 registers per thread. The number of 32-bit registers required for the blocking is calculated as follows assuming the mma instruction is used and each element in a matrix is stored by only one element of a fragment without duplication.

$$n_{\text{Regs}} = ((\underbrace{n^2}_{A_{\text{reg}}} + \underbrace{n^2}_{B_{\text{reg}}} + \underbrace{n^2}_{C_{\text{reg}}}) \times \frac{1}{2}) / \text{warpSize} = \frac{1}{16}n^2. \quad (2)$$

For instance, in the case of  $n = 64$ , the number of required registers is 256, and the registers spill to local memory. Therefore, we need to reduce the shared memory access not by increasing the register blocking size. Furthermore, the Tensor Cores performance has been improved more than the shared memory bandwidth on NVIDIA A100 compared to



**Figure 3.** The arithmetic intensity of matrix-matrix multiplication for each size of register blocking blocking- $(n, n, n)$ .

V100. This can be seen from the fact that the AI value at the boundary between the memory bandwidth and the computational performance bound is smaller for A100 than for V100.

## 4 WMMA API extension library

To leverage the high Tensor Cores performance, it is necessary to supply matrix data to Tensor Core with sufficient throughput. However, due to the limited functionality of the WMMA API, the throughput improvements that can be made using only the WMMA API are limited. Therefore, we implement a WMMA API Extension library (WMMAe) to reinforce the functionality of WMMA API. The WMMAe consists of the following two components:

1. Primitive functions
2. SGEMM emulation on Tensor Cores using Error Correction method (WMMAe-TCEC)

In this section, we show the functionality of these components and evaluate the performance improvement compared to only using WMMA API. We use NVIDIA A100 40GB SXM4 and NVIDIA V100 16GB PCIe GPUs for the evaluations.

### 4.1 Primitive functions

We can generate a fragment of a matrix in which all elements are the same value without shared memory access using `fill_fragment` function in WMMA API. On the other hand, to generate fragments of other matrices, it is necessary to explicitly store the matrix in shared memory and load it using `load_matrix_sync` function in WMMA API. Now, we consider the matrices that have some structural rules. For instance, when performing scan operations using matrix-vector multiplication, we need an upper triangular matrix  $\mathbf{U}$  in which all non-zero elements are one. Then, we perform a scan operation to an array  $[a_0 \ a_1 \ \dots \ a_{n-1}]$  using  $n \times n$  matrix  $\mathbf{U}$  as follows:

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}^\top \cdot \mathbf{U} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}^\top \cdot \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & 1 & \cdots & 1 \\ 0 & 0 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} = \begin{bmatrix} a_0 \\ \sum_{i=0}^1 a_i \\ \vdots \\ \sum_{i=0}^{n-1} a_i \end{bmatrix}^\top.$$

The structural rule for the  $(i, j)$  element of the matrix  $\mathbf{U}$  is as follows:

$$u_{i,j} = \begin{cases} 1 & i \leq j \\ 0 & \text{Otherwise} \end{cases} \quad (3)$$

Dokkak *et al.* utilize the rule for generating the fragment of the matrix without storing it explicitly in shared memory. We generalize the functionality and provide functions for generating a fragment of any matrix from its structural rule: `foreach_ij` and `map`.

### 4.2 Primitive function : `foreach_ij`

The `foreach_ij` function calculates the mapping between matrix element position  $(i, j)$  and fragment indices and gives them to a given lambda function. In the lambda function, we calculate the value of the  $(i, j)$  element of the matrix and set it to the fragment using the given mapping information. For instance, we show a pseudocode for generating the matrix  $\mathbf{U}$  fragment by the rule in Eq (3) in Code 3. Strictly speaking, since one element in a matrix is kept by two fragment elements when using WMMA API for C++, `foreach_ij` function gives the list of fragment element indices to the lambda function. However, in this pseudocode, we simplify the argument of the lambda function as only one fragment index is given. By using this function, we can generate a fragment of any matrix from its structural rule without storing it in shared memory.

```

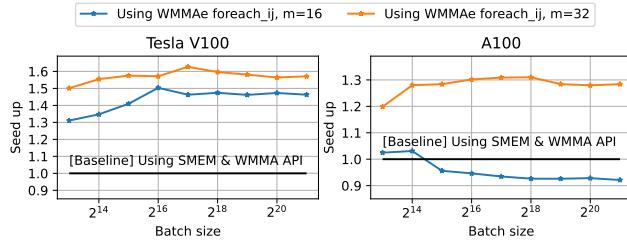
1 fragment <16, 16, 16> frag;
2 foreach_ij <decltype(frag)>(
3     // The lambda function to set each fragment
4     elements
5     [&](fid, i, j) {
6         if (i <= j) frag.x[fid] = 1;
7         else frag.x[fid] = 0;
8     });

```

**Code 3.** Generating the matrix  $\mathbf{U}$  fragment from the structural rule in Eq. (3) using WMMAe `foreach_ij` function.

**4.2.1 Performance evaluation.** We use a batched Householder transformation benchmark for evaluating the performance improvement by `foreach_ij` function. The Householder transformation is one of the orthogonal transformations used for QR factorization etc. This transformation is calculated as follows for a  $n \times n$  Householder matrix  $\mathbf{H}$ ,  $m \times k$  input matrix  $\mathbf{A}$ :

$$\mathbf{H} \cdot \mathbf{A} = (\mathbf{I}_m - 2\mathbf{v}^\top \mathbf{v}) \cdot \mathbf{A}, \quad (4)$$



**Figure 4.** The performance evaluation of `foreach_ij` function using batched Householder benchmark, where we multiply a  $m \times m$  Householder matrix  $\mathbf{H}$  with an input matrix  $\mathbf{A}$  using Tensor Cores.

where  $\mathbf{v}$  is a  $m$ -dimensional identity vector and  $\mathbf{I}_m$  is a  $m \times m$  identity matrix. In this benchmark, we explicitly compute the Householder matrix  $\mathbf{H}$  from  $\mathbf{v}$  and multiply it by  $\mathbf{A}$ . This computation is performed for  $b$  (batch size) FP16 input matrices  $\mathbf{A}_i$  and FP16 vectors  $\mathbf{v}_i$ . To obtain the baseline performance, we implemented the batched Householder transformation, which stores the Householder matrix in shared memory and loads it using the WMMA API function. Then the multiplication of  $\mathbf{A}$  and  $\mathbf{H}$  is performed on Tensor Cores. We show a speed-up ratio using WMMAe in Figure 4. We can see that the performance is improved using `foreach_ij` on V100 GPU in both cases. On the other hand, for  $m = 16$  on A100, the implementation using `foreach_ij` has a lower performance compared to the baseline. In this case, the pseudocode of the implementation is shown in Code 4.

```

1 fragment <16, 16, 16> frag ;
2 foreach_ij < decltype(frag) >(
3     [&](fid, i, j) {
4         auto elm = v[i]*v[j]*(-2);
5         if (i==j) elm += 1;
6         frag.x[fid] = elm;
7     });

```

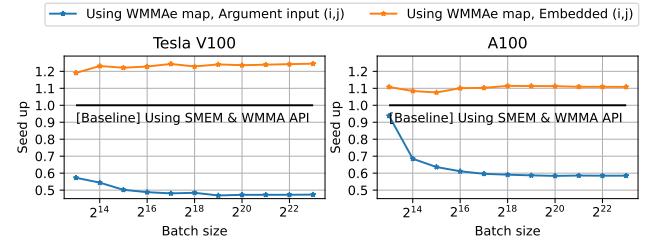
**Code 4.** Generating a  $16 \times 16$  Householder matrix fragment using `foreach_ij`.

In this code, the cost of the mapping calculation is higher than the cost of storing the matrix explicitly in shared memory, which might be the reason for the low performance. Whereas, for  $m = 32$  on A100, the implementation using `foreach_ij` has higher performance than the baseline. In this case, the pseudocode of the implementation is shown in Code 5.

```

1 fragment <16, 16, 16> frag[2 * 2]; // 32x32 matrix
2 foreach_ij < decltype(*frag) >(
3     [&](fid, i, j) {
4         for (unsigned bi = 0; bi < 2; bi++) {
5             for (unsigned bj = 0; bj < 2; bj++) {
6                 auto elm = v[i+bi*16] * v[j+bj*16]*(-2);
7                 if (i==j) elm += 1;
8                 frag[bi+bj*2].x[fid] = elm;
9             }
10        }
11    });

```



**Figure 5.** The performance evaluation of `map` function using batched Given's rotation benchmark. The "Argument input (i,j)" means that the parameter (i, j) for Given's rotation matrix is set through kernel function arguments, and "Embedded (i,j)" means that these parameters are set in compile-time.

```
9     }}}} );
```

**Code 5.** Generating an array of fragments for  $32 \times 32$  Householder matrix using `foreach_ij`.

For the  $32 \times 32$  matrix fragment, we used a  $2 \times 2$  array of fragments holding matrices of size  $16 \times 16$ . The elements of all the fragments are set in a single `foreach_ij` function. This means that four fragments are generated in one mapping calculation, and the cost of the mapping calculation is relatively lower than that of the  $m = 16$  case. Thus, we consider that reusing the mapping calculation among several fragments is important to speed up the use of the `foreach_ij` function.

### 4.3 Primitive function : map

The map function takes the position  $(i, j)$  of an element of the matrix as an argument and returns a pair  $(\text{lid}, \text{fid})$  of the thread number (lane id; lid) in a warp and the element number of the fragment holding this element. Using this function, we can manipulate any  $(i, j)$  element of the matrix as a fragment. For instance, Code 6 sets the  $(i, j)$  element of a matrix  $\mathbf{A}$ , which is held as a fragment, to 1.

```

1 fragment frag_a;
2 unsigned lid, fid;
3 // Calculate lid and fid from matrix position (i,
4 // j)
4 map< decltype(frag) >(lid, fid /*=2*/ , i, j);
5 // Set 1
6 if ((threadIdx.x & 0x1f) == lid) {
7     frag_a.x[fid] = 1;
8 }

```

**Code 6.** Setting  $(i, j)$ -element of a matrix held as fragment using WMMAe map function.

**4.3.1 Performance evaluation.** We define a batched Given's rotation benchmark to evaluate the performance improvement by the map function. The Given's rotation is a rotation operation for a vector and matrix and is used for QR factorization etc. The definition of Given's rotation for a matrix  $\mathbf{A}$

is as follows:

$$G(i, j, \theta) \cdot A, \quad (5)$$

where

$$G(i, j, \theta) = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & c & \cdots & -s & \cdots \\ & \vdots & \ddots & \vdots & \\ s & \cdots & c & \cdots & \\ & \vdots & & \ddots & \\ & & \vdots & & \\ & & & & 1 \end{bmatrix}_{i\text{-th} \times j\text{-th}}$$

$c = \cos \theta$  and  $s = \sin \theta$ . In this benchmark, Given's rotation operations for  $b$  FP16 input matrices  $A_k$  are performed by multiplying by  $G(i, j, \theta_k)$  in parallel. The  $i$  and  $j$  are fixed in all calculations, and there are two ways to fix them as follows: 1) Specify them as arguments of the kernel function. 2) Embed them in the kernel function. When generating a fragment of matrix  $G$  using the map function, first, all elements in the fragment are filled with zeros by WMMA API `fill_fragment` function. Then set 1,  $s$ , and  $c$  at each position using the map function. To obtain the baseline performance, the matrix  $G$  is explicitly stored in shared memory and loaded using a WMMA API function. We show a speed-up ratio by the mapping function in Figure 5. When  $i$  and  $j$  are given as arguments of the kernel function, it is slower than the baseline implementation. On the other hand, when  $i$  and  $j$  are embedded in the kernel function, then the baseline implementation. When  $i$  and  $j$  are embedded in the kernel function, compiler optimization reduces the computing amount of mapping calculation and required registers at runtime.

#### 4.4 WMMAe-TCEC

When computing single-precision matrix-matrix multiplication on Tensor Cores, we need to convert input matrices to FP16 ones. This conversion results in a loss of accuracy in the resulting matrix. Markidis *et al.* [10] proposed a method for single-precision matrix multiplication using Tensor Cores with error correction. However, the accuracy of their method does not match the single-precision. In our previous research, we improve the accuracy and reduce the computation complexity of their method [11]. In our method, they compute the single-precision matrix-matrix multiplication  $C_{F32} = A_{F32}B_{F32}$  as follows:

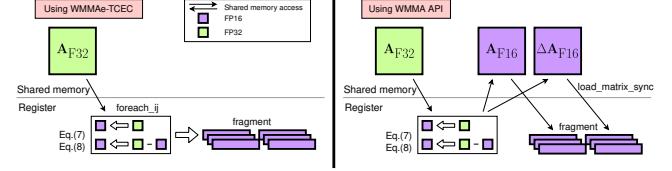
$$A_{F16} \leftarrow \text{toFP16}(A_{F32}) \quad (6)$$

$$\Delta A_{F16} \leftarrow \text{toFP16}((A_{F32} - \text{toFP32}(A_{F16})) \times 2^{11})$$

$$B_{F16} \leftarrow \text{toFP16}(B_{F32}) \quad (7)$$

$$\Delta B_{F16} \leftarrow \text{toFP16}((B_{F32} - \text{toFP32}(B_{F16})) \times 2^{11})$$

$$C_{F32} \leftarrow A_{F16}B_{F16} + (\Delta A_{F16}B_{F16} + A_{F16}\Delta B_{F16}) / 2^{11},$$



**Figure 6.** The comparison of data flow between using WMMA API and WMMAe. Here we load fragments for SGEMM emulation on Tensor Cores using error correction without additional shared memories  $A_{F16}$  and  $\Delta A_{F16}$ , that are required when using WMMA API.

where `toFP16` and `toFP32` are the conversion to FP16 and FP32, respectively. We improve the matrix-matrix multiplication accuracy by avoiding the rounding inside Tensor Cores, RZ, and achieve the same accuracy with FP32 SIMT Core computation. Although we have included our method in NVIDIA CUTLASS and evaluated the accuracy, performance, and power consumption in the previous paper, the matrix-matrix multiplication is inside various linear algebra algorithms, and we would like to use the computation inside custom kernel functions. Therefore, we provide functionality for using this method inside a custom kernel function.

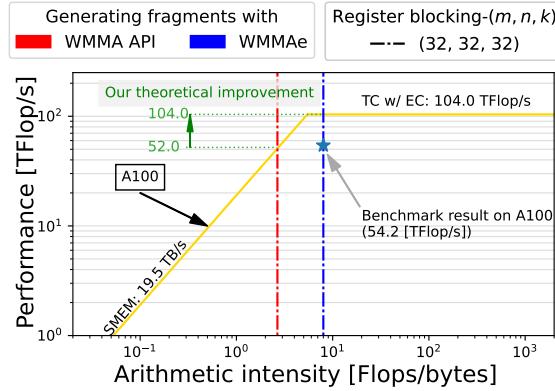
To compute the Eqs. (6)-(8) using WMMA API for C++, we need to store the matrices  $A_{F16}$ ,  $\Delta A_{F16}$  in the shared memory explicitly since the mapping function `load_matrix_sync` in WMMA API only makes the fragment from memory as shown in the top of Figure 6. On the other hand, we can avoid the explicit storing by `foreach_ij` function in WMMAe. Using this function, we implement WMMAe-TCEC, which reduces the memory footprint and provides the error correction computation with the same interface as WMMA API. The WMMAe-TCEC includes a function for generating the fragments of  $A_{F16}$  and  $\Delta A_{F16}$  directly from the input matrix  $A_{F32}$  shown in the bottom of Figure 6. We can use WMMAe-TCEC just by changing the matrix data types and the namespace in Code 1 from `nvcuda::wmma` to `mtk::wmma::tcec`.

Moreover, since the WMMAe-TCEC adopts a policy-based design, we can change the following backward computation by only changing the policy, which is specified as an optional template parameter of the fragment.

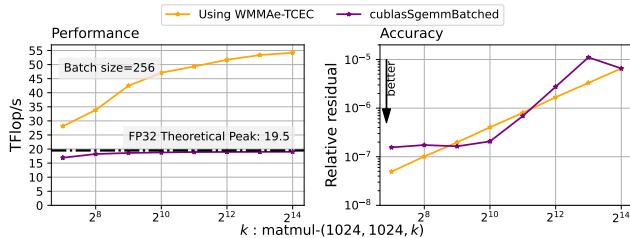
- Tensor Core instruction: Use the `wmma` instructions or `mma` instruction.
- Error correction: Enable or disable.
- Use Tensor Core or software systolic array [1].

Using this feature, we can evaluate the effect of the error correction method easily.

**4.4.1 Theoretical performance analysis.** We show the AI of matrix-matrix multiplication with error correction that we used for the performance evaluation in Figure 7. By using WMMAe-TCEC, we can increase the AI and improve the theoretical computing performance bounded by the shared



**Figure 7.** The arithmetic intensity of SGEMM emulation on Tensor Cores using error correction method. The peak performance is calculated by dividing the theoretical peak performance of FP16-TC in Table 1 by 3 since we need 3 times matrix-matrix multiplication in Eq. (8).



**Figure 8.** The throughput and accuracy evaluation of batched SGEMM using WMMAe-TCEC.

memory bandwidth. Although we can increase the AI by increasing the size of register blocking, the number of registers that one thread can use is limited by the hardware. For instance, in the case of  $(m, n, k) = (32, 32, 32)$ , which is used in our benchmark evaluation, we need 128 32-bit registers to keep the fragments, which amounts to 50% of registers that one thread can use. The registers are used not only for fragments but also for memory access offset calculations and other floating-point value operations such as eq. (7). Reducing the number of required registers can improve the throughput since it can improve occupancy. And when the number of required registers exceeds the hardware limitation, the device memory is used instead, which results in performance degradation. Therefore, increasing the AI without increasing the register blocking size is advantageous.

**4.4.2 Performance evaluation.** We use a batched matrix-matrix multiplication benchmark to evaluate the performance and accuracy of the WMMAe-TCEC. In this benchmark, we compute 256 matrix-matrix multiplications  $\mathbf{A}_i \cdot \mathbf{B}_i$  where each  $\mathbf{A}_i$  and  $\mathbf{B}_i$  are  $1024 \times k$  and  $k \times 1024$  FP32 matrices. Then, we calculate the computing performance from

the computing time  $t$  [s] as  $(2 \times 1024 \times 1024 \times k/t)$  [Flop/s], and a max relative error for the accuracy. We show the performance and accuracy comparison between our implementation using WMMAe-TCEC and cuBLAS batched SGEMM function in Figure 8. In our implementation, we use the mma instruction, and the shared memory and register blocking sizes are  $(128, 128, 32)$  and  $(32, 32, 32)$ , respectively. We found this blocking size using a grid search that experimentally maximizes the throughput on NVIDIA A100 (40GB, SXM4) GPU. The outcome of our evaluation shows that our implementation achieves 54.2 [TFlop/s], which outperforms the theoretical peak performance of FP32 on NVIDIA A100, while the accuracy remains the same with cuBLAS SGEMM. The achieved throughput is larger than the throughput of SGEMM emulation that we have achieved using the NVIDIA CUTLASS library (51 TFlop/s) in our previous paper [11]. According to the roofline model, when we only use WMMA API, the theoretical peak performance for our chosen register blocking size is limited to 52.0 TFlop/s bounded by the shared memory bandwidth. Therefore, the achieved throughput can not be achieved without reducing the shared memory footprint that our library does. However, by using WMMAe, we improved the theoretical peak performance of this method to 104.0 TFlop/s by reducing the shared memory footprint. Since the achieved efficiency is only 52% of the theoretical peak performance, we believe there is room for improving the throughput.

We summarize the advantages of WMMAe-TCEC as follows:

- It provides an interface for the single-precision emulation method on Tensor Cores, which has the same interface as NVIDIA WMMA API.
- It improves the theoretical peak performance of matrix-matrix multiplication with error correction by reducing shared memory footprint without increasing register usage.
- It reduces the shared memory usage required to store the fragments of FP16 matrices when using only WMMA API.
- It is proved to outperform the FP32 theoretical peak performance on NVIDIA A100 experimentally while the accuracy remains the same with FP32 computation.

## 5 Conclusion

We have investigated a simple matrix-matrix multiplication on Tensor Cores by roofline model and found that reducing the shared memory footprint is necessary to fully exploit the high throughput of Tensor Cores. To reduce the footprint, we implement a WMMA API extension library which allows us to generate fragments flexibly. This library is open-source and available on GitHub. We show that this library can improve the computing throughput on Tensor Cores. Furthermore, we improve the theoretical peak performance

of single precision matrix-matrix multiplication emulation on Tensor Cores, which is bounded by the shared memory bandwidth when using only WMMA API. Then, we provide this functionality with the same interface as WMMA API. We also show that this functionality can outperform the FP32 theoretical peak performance on NVIDIA A100 GPU. We believe such a faster data supply is necessary to maximize the use of high-speed matrix multiplication units in future architectures.

## Acknowledgments

This work was partially supported by JSPS KAKENHI JP22H03598 and JP21J14694. This work was partially supported by "Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures" in Japan (Project ID: jh220022-NAHI)

## References

- [1] Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. 2019. A versatile software systolic execution model for GPU memory-bound kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*. Association for Computing Machinery, New York, NY, USA, 1–81. <https://doi.org/10.1145/3295500.3356162>
- [2] NVIDIA Corporation. 2022. NVIDIA H100 TENSOR CORE GPU. <https://resources.nvidia.com/en-us-tensor-core/nvidia-h100-datasheet>
- [3] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*. Association for Computing Machinery, New York, NY, USA, 46–57. <https://doi.org/10.1145/3330345.3331057>
- [4] Joshua Finkelstein, Emanuel H. Rubensson, Susan M. Mniszewski, Christian F. A. Negre, and Anders M. N. Niklasson. 2022. Quantum Perturbation Theory Using Tensor Cores and a Deep Neural Network. *Journal of Chemical Theory and Computation* 18, 7 (July 2022), 4255–4268. <https://doi.org/10.1021/acs.jctc.2c00274> Publisher: American Chemical Society.
- [5] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham. 2018. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 603–613. <https://doi.org/10.1109/SC.2018.00050>
- [6] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv:1804.06826 [cs]* (April 2018). <http://arxiv.org/abs/1804.06826> arXiv: 1804.06826
- [7] Norman P. Jouppi, Cliff Young, Nishant Patil, and et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. <https://doi.org/10.1145/3079856.3080246>
- [8] Binrui Li, Shenggan Cheng, and James Lin. 2021. tcFFT: Accelerating Half-Precision FFT through Tensor Cores. *arXiv:2104.11471 [cs]* (April 2021). <http://arxiv.org/abs/2104.11471> arXiv: 2104.11471 version: 1.
- [9] Junichiro Makino. 2021. “Near-Optimal” Designs. In *Principles of High-Performance Processor Design: For High Performance Computing, Deep Neural Networks and Data Science*, Junichiro Makino (Ed.). Springer International Publishing, Cham, 95–134. [https://doi.org/10.1007/978-3-030-76871-3\\_5](https://doi.org/10.1007/978-3-030-76871-3_5)
- [10] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (May 2018), 522–531. <https://doi.org/10.1109/IPDPSW.2018.00091> arXiv: 1803.04014.
- [11] Hiroyuki Ootomo and Rio Yokota. 2022. Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance. *The International Journal of High Performance Computing Applications* (June 2022). <https://doi.org/10.1177/10943420221090256> Publisher: SAGE PublicationsSage UK: London, England.
- [12] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. <https://doi.org/10.1145/1498765.1498785>

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

---

# RoFORMER: ENHANCED TRANSFORMER WITH ROTARY POSITION EMBEDDING

---

**Jianlin Su**

Zhuiyi Technology Co., Ltd.  
Shenzhen  
bojonesu@wezhuiyi.com

**Yu Lu**

Zhuiyi Technology Co., Ltd.  
Shenzhen  
julianlu@wezhuiyi.com

**Shengfeng Pan**

Zhuiyi Technology Co., Ltd.  
Shenzhen  
nickpan@wezhuiyi.com

**Ahmed Murtadha**

Zhuiyi Technology Co., Ltd.  
Shenzhen  
mengjiayi@wezhuiyi.com

**Bo Wen**

Zhuiyi Technology Co., Ltd.  
Shenzhen  
brucewen@wezhuiyi.com

**Yunfeng Liu**

Zhuiyi Technology Co., Ltd.  
Shenzhen  
glenliu@wezhuiyi.com

August 10, 2022

## ABSTRACT

Position encoding recently has shown effective in the transformer architecture. It enables valuable supervision for dependency modeling between elements at different positions of the sequence. In this paper, we first investigate various methods to integrate positional information into the learning process of transformer-based language models. Then, we propose a novel method named Rotary Position Embedding(RoPE) to effectively leverage the positional information. Specifically, the proposed RoPE encodes the absolute position with a rotation matrix and meanwhile incorporates the explicit relative position dependency in self-attention formulation. Notably, RoPE enables valuable properties, including the flexibility of sequence length, decaying inter-token dependency with increasing relative distances, and the capability of equipping the linear self-attention with relative position encoding. Finally, we evaluate the enhanced transformer with rotary position embedding, also called RoFormer, on various long text classification benchmark datasets. Our experiments show that it consistently overcomes its alternatives. Furthermore, we provide a theoretical analysis to explain some experimental results. RoFormer is already integrated into Huggingface: [https://huggingface.co/docs/transformers/model\\_doc/roformer](https://huggingface.co/docs/transformers/model_doc/roformer).

**Keywords** Pre-trained Language Models · Position Information Encoding · Pre-training · Natural Language Processing.

## 1 Introduction

The sequential order of words is of great value to natural language understanding. Recurrent neural networks (RNNs) based models encode tokens' order by recursively computing a hidden state along the time dimension. Convolution neural networks (CNNs) based models (CNNs) Gehring et al. [2017] were typically considered position-agnostic, but recent work Islam et al. [2020] has shown that the commonly used padding operation can implicitly learn position information. Recently, the pre-trained language models (PLMs), which were built upon the transformer Vaswani et al. [2017], have achieved the state-of-the-art performance of various natural language processing (NLP) tasks, including context representation learning Devlin et al. [2019], machine translation Vaswani et al. [2017], and language modeling Radford et al. [2019], to name a few. Unlike, RNNs and CNNs-based models, PLMs utilize the self-attention mechanism to semantically capture the contextual representation of a given corpus. As a consequence, PLMs achieve a significant improvement in terms of parallelization over RNNs and improve the modeling ability of longer intra-token relations compared to CNNs<sup>1</sup>.

<sup>1</sup>A stack of multiple CNN layers can also capture longer intra-token relation, here we only consider single layer setting.

It is noteworthy that the self-attention architecture of the current PLMs has shown to be position-agnostic Yun et al. [2020]. Following this claim, various approaches have been proposed to encode the position information into the learning process. On one side, generated absolute position encoding through a pre-defined function Vaswani et al. [2017] was added to the contextual representations, while a trainable absolute position encoding Gehring et al. [2017], Devlin et al. [2019], Lan et al. [2020], Clark et al. [2020], Radford et al. [2019], Radford and Narasimhan [2018]. On the other side, the previous work Parikh et al. [2016], Shaw et al. [2018], Huang et al. [2018], Dai et al. [2019], Yang et al. [2019], Raffel et al. [2020], Ke et al. [2020], He et al. [2020], Huang et al. [2020] focuses on relative position encoding, which typically encodes the relative position information into the attention mechanism. In addition to these approaches, the authors of Liu et al. [2020] have proposed to model the dependency of position encoding from the perspective of Neural ODE Chen et al. [2018a], and the authors of Wang et al. [2020] have proposed to model the position information in complex space. Despite the effectiveness of these approaches, they commonly add the position information to the context representation and thus render them unsuitable for the linear self-attention architecture.

In this paper, we introduce a novel method, namely Rotary Position Embedding(RoPE), to leverage the positional information into the learning process of PLMS. Specifically, RoPE encodes the absolute position with a rotation matrix and meanwhile incorporates the explicit relative position dependency in self-attention formulation. Note that the proposed RoPE is prioritized over the existing methods through valuable properties, including the sequence length flexibility, decaying inter-token dependency with increasing relative distances, and the capability of equipping the linear self-attention with relative position encoding. Experimental results on various long text classification benchmark datasets show that the enhanced transformer with rotary position embedding, namely RoFormer, can give better performance compared to baseline alternatives and thus demonstrates the efficacy of the proposed RoPE.

In brief, our contributions are three-folds as follows:

- We investigated the existing approaches to the relative position encoding and found that they are mostly built based on the idea of the decomposition of adding position encoding to the context representations. We introduce a novel method, namely Rotary Position Embedding(RoPE), to leverage the positional information into the learning process of PLMS. The key idea is to encode relative position by multiplying the context representations with a rotation matrix with a clear theoretical interpretation.
- We study the properties of RoPE and show that it decays with the relative distance increased, which is desired for natural language encoding. We kindly argue that previous relative position encoding-based approaches are not compatible with linear self-attention.
- We evaluate the proposed RoFormer on various long text benchmark datasets. Our experiments show that it consistently achieves better performance compared to its alternatives. Some experiments with pre-trained language models are available on GitHub: <https://github.com/ZhuiyiTechnology/roformer>.

The remaining of the paper is organized as follows. We establish a formal description of the position encoding problem in self-attention architecture and revisit previous works in Section (2). We then describe the rotary position encoding (RoPE) and study its properties in Section (3). We report experiments in Section (4). Finally, we conclude this paper in Section (5).

## 2 Background and Related Work

### 2.1 Preliminary

Let  $\mathbb{S}_N = \{w_i\}_{i=1}^N$  be a sequence of  $N$  input tokens with  $w_i$  being the  $i^{th}$  element. The corresponding word embedding of  $\mathbb{S}_N$  is denoted as  $\mathbb{E}_N = \{\mathbf{x}_i\}_{i=1}^N$ , where  $\mathbf{x}_i \in \mathbb{R}^d$  is the d-dimensional word embedding vector of token  $w_i$  without position information. The self-attention first incorporates position information to the word embeddings and transforms them into queries, keys, and value representations.

$$\begin{aligned} \mathbf{q}_m &= f_q(\mathbf{x}_m, m) \\ \mathbf{k}_n &= f_k(\mathbf{x}_n, n) \\ \mathbf{v}_n &= f_v(\mathbf{x}_n, n), \end{aligned} \tag{1}$$

where  $\mathbf{q}_m$ ,  $\mathbf{k}_n$  and  $\mathbf{v}_n$  incorporate the  $m^{th}$  and  $n^{th}$  positions through  $f_q$ ,  $f_k$  and  $f_v$ , respectively. The query and key values are then used to compute the attention weights, while the output is computed as the weighted sum over the value

representation.

$$\begin{aligned} a_{m,n} &= \frac{\exp(\frac{\mathbf{q}_m^\top \mathbf{k}_n}{\sqrt{d}})}{\sum_{j=1}^N \exp(\frac{\mathbf{q}_m^\top \mathbf{k}_j}{\sqrt{d}})} \\ \mathbf{o}_m &= \sum_{n=1}^N a_{m,n} \mathbf{v}_n \end{aligned} \quad (2)$$

The existing approaches of transformer-based position encoding mainly focus on choosing a suitable function to form Equation (1).

## 2.2 Absolute position embedding

A typical choice of Equation (1) is

$$f_{t:t \in \{q,k,v\}}(\mathbf{x}_i, i) := \mathbf{W}_{t:t \in \{q,k,v\}}(\mathbf{x}_i + \mathbf{p}_i), \quad (3)$$

where  $\mathbf{p}_i \in \mathbb{R}^d$  is a d-dimensional vector depending of the position of token  $\mathbf{x}_i$ . Previous work Devlin et al. [2019], Lan et al. [2020], Clark et al. [2020], Radford et al. [2019], Radford and Narasimhan [2018] introduced the use of a set of trainable vectors  $\mathbf{p}_i \in \{\mathbf{p}_t\}_{t=1}^L$ , where  $L$  is the maximum sequence length. The authors of Vaswani et al. [2017] have proposed to generate  $\mathbf{p}_i$  using the sinusoidal function.

$$\begin{cases} \mathbf{p}_{i,2t} &= \sin(k/10000^{2t/d}) \\ \mathbf{p}_{i,2t+1} &= \cos(k/10000^{2t/d}) \end{cases} \quad (4)$$

in which  $\mathbf{p}_{i,2t}$  is the  $2t^{th}$  element of the d-dimensional vector  $\mathbf{p}_i$ . In the next section, we show that our proposed RoPE is related to this intuition from the sinusoidal function perspective. However, instead of directly adding the position to the context representation, RoPE proposes to incorporate the relative position information by multiplying with the sinusoidal functions.

## 2.3 Relative position embedding

The authors of Shaw et al. [2018] applied different settings of Equation (1) as following:

$$\begin{aligned} f_q(\mathbf{x}_m) &:= \mathbf{W}_q \mathbf{x}_m \\ f_k(\mathbf{x}_n, n) &:= \mathbf{W}_k(\mathbf{x}_n + \tilde{\mathbf{p}}_r^k) \\ f_v(\mathbf{x}_n, n) &:= \mathbf{W}_v(\mathbf{x}_n + \tilde{\mathbf{p}}_r^v) \end{aligned} \quad (5)$$

where  $\tilde{\mathbf{p}}_r^k, \tilde{\mathbf{p}}_r^v \in \mathbb{R}^d$  are trainable relative position embeddings. Note that  $r = \text{clip}(m - n, r_{\min}, r_{\max})$  represents the relative distance between position  $m$  and  $n$ . They clipped the relative distance with the hypothesis that precise relative position information is not useful beyond a certain distance. Keeping the form of Equation (3), the authors Dai et al. [2019] have proposed to decompose  $\mathbf{q}_m^\top \mathbf{k}_n$  of Equation (2) as

$$\mathbf{q}_m^\top \mathbf{k}_n = \mathbf{x}_m^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{x}_n + \mathbf{x}_m^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{p}_n + \mathbf{p}_m^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{x}_n + \mathbf{p}_m^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{p}_n, \quad (6)$$

the key idea is to replace the absolute position embedding  $\mathbf{p}_n$  with its sinusoid-encoded relative counterpart  $\tilde{\mathbf{p}}_{m-n}$ , while the absolute position  $\mathbf{p}_m$  in the third and fourth term with two trainable vectors  $\mathbf{u}$  and  $\mathbf{v}$  independent of the query positions. Further,  $\mathbf{W}_k$  is distinguished for the content-based and location-based key vectors  $\mathbf{x}_n$  and  $\mathbf{p}_n$ , denoted as  $\mathbf{W}_k$  and  $\tilde{\mathbf{W}}_k$ , resulting in:

$$\mathbf{q}_m^\top \mathbf{k}_n = \mathbf{x}_m^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{x}_n + \mathbf{x}_m^\top \mathbf{W}_q^\top \tilde{\mathbf{W}}_k \tilde{\mathbf{p}}_{m-n} + \mathbf{u}^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{x}_n + \mathbf{v}^\top \mathbf{W}_q^\top \tilde{\mathbf{W}}_k \tilde{\mathbf{p}}_{m-n} \quad (7)$$

It is noteworthy that the position information in the value term is removed by setting  $f_v(\mathbf{x}_j) := \mathbf{W}_v \mathbf{x}_j$ . Later work Raffel et al. [2020], He et al. [2020], Ke et al. [2020], Huang et al. [2020] followed these settings by only encoding the relative position information into the attention weights. However, the authors of Raffel et al. [2020] reformed Equation (6) as:

$$\mathbf{q}_m^\top \mathbf{k}_n = \mathbf{x}_m^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{x}_n + b_{i,j} \quad (8)$$

where  $b_{i,j}$  is a trainable bias. The authors of Ke et al. [2020] investigated the middle two terms of Equation (6) and found little correlations between absolute positions and words. The authors of Raffel et al. [2020] proposed to model a pair of words or positions using different projection matrices.

$$\mathbf{q}_m^\top \mathbf{k}_n = \mathbf{x}_m^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{x}_n + \mathbf{p}_m^\top \mathbf{U}_q^\top \mathbf{U}_k \mathbf{p}_n + b_{i,j} \quad (9)$$

The authors of He et al. [2020] argued that the relative positions of two tokens could only be fully modeled using the middle two terms of Equation (6). As a consequence, the absolute position embeddings  $\mathbf{p}_m$  and  $\mathbf{p}_n$  were simply replaced with the relative position embeddings  $\tilde{\mathbf{p}}_{m-n}$ :

$$\mathbf{q}_m^\top \mathbf{k}_n = \mathbf{x}_m^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{x}_n + \mathbf{x}_m^\top \mathbf{W}_q^\top \mathbf{W}_k \tilde{\mathbf{p}}_{m-n} + \tilde{\mathbf{p}}_{m-n}^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{x}_n \quad (10)$$

A comparison of the four variants of the relative position embeddings Radford and Narasimhan [2018] has shown that the variant similar to Equation (10) is the most efficient among the other three. Generally speaking, all these approaches attempt to modify Equation (6) based on the decomposition of Equation (3) under the self-attention settings in Equation (2), which was originally proposed in Vaswani et al. [2017]. They commonly introduced to directly add the position information to the context representations. Unlikely, our approach aims to derive the relative position encoding from Equation (1) under some constraints. Next, we show that the derived approach is more interpretable by incorporating relative position information with the rotation of context representations.

### 3 Proposed approach

In this section, we discuss the proposed rotary position embedding (RoPE). We first formulate the relative position encoding problem in Section (3.1), we then derive the RoPE in Section (3.2) and investigate its properties in Section (3.3).

#### 3.1 Formulation

Transformer-based language modeling usually leverages the position information of individual tokens through a self-attention mechanism. As can be observed in Equation (2),  $\mathbf{q}_m^\top \mathbf{k}_n$  typically enables knowledge conveyance between tokens at different positions. In order to incorporate relative position information, we require the inner product of query  $\mathbf{q}_m$  and key  $\mathbf{k}_n$  to be formulated by a function  $g$ , which takes only the word embeddings  $\mathbf{x}_m$ ,  $\mathbf{x}_n$ , and their relative position  $m - n$  as input variables. In other words, we hope that the inner product encodes position information only in the relative form:

$$\langle f_q(\mathbf{x}_m, m), f_k(\mathbf{x}_n, n) \rangle = g(\mathbf{x}_m, \mathbf{x}_n, m - n). \quad (11)$$

The ultimate goal is to find an equivalent encoding mechanism to solve the functions  $f_q(\mathbf{x}_m, m)$  and  $f_k(\mathbf{x}_n, n)$  to conform the aforementioned relation.

#### 3.2 Rotary position embedding

##### 3.2.1 A 2D case

We begin with a simple case with a dimension  $d = 2$ . Under these settings, we make use of the geometric property of vectors on a 2D plane and its complex form to prove (refer Section (3.4.1) for more details) that a solution to our formulation Equation (11) is:

$$\begin{aligned} f_q(\mathbf{x}_m, m) &= (\mathbf{W}_q \mathbf{x}_m) e^{im\theta} \\ f_k(\mathbf{x}_n, n) &= (\mathbf{W}_k \mathbf{x}_n) e^{in\theta} \\ g(\mathbf{x}_m, \mathbf{x}_n, m - n) &= \text{Re}[(\mathbf{W}_q \mathbf{x}_m)(\mathbf{W}_k \mathbf{x}_n)^* e^{i(m-n)\theta}] \end{aligned} \quad (12)$$

where  $\text{Re}[\cdot]$  is the real part of a complex number and  $(\mathbf{W}_k \mathbf{x}_n)^*$  represents the conjugate complex number of  $(\mathbf{W}_k \mathbf{x}_n)$ .  $\theta \in \mathbb{R}$  is a preset non-zero constant. We can further write  $f_{\{q,k\}}$  in a multiplication matrix:

$$f_{\{q,k\}}(\mathbf{x}_m, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} W_{\{q,k\}}^{(11)} & W_{\{q,k\}}^{(12)} \\ W_{\{q,k\}}^{(21)} & W_{\{q,k\}}^{(22)} \end{pmatrix} \begin{pmatrix} x_m^{(1)} \\ x_m^{(2)} \end{pmatrix} \quad (13)$$

where  $(x_m^{(1)}, x_m^{(2)})$  is  $\mathbf{x}_m$  expressed in the 2D coordinates. Similarly,  $g$  can be viewed as a matrix and thus enables the solution of formulation in Section (3.1) under the 2D case. Specifically, incorporating the relative position embedding is straightforward: simply rotate the affine-transformed word embedding vector by amount of angle multiples of its position index and thus interprets the intuition behind *Rotary Position Embedding*.

### 3.2.2 General form

In order to generalize our results in 2D to any  $\mathbf{x}_i \in \mathbb{R}^d$  where  $d$  is even, we divide the  $d$ -dimension space into  $d/2$  sub-spaces and combine them in the merit of the linearity of the inner product, turning  $f_{\{q,k\}}$  into:

$$f_{\{q,k\}}(\mathbf{x}_m, m) = \mathbf{R}_{\Theta,m}^d \mathbf{W}_{\{q,k\}} \mathbf{x}_m \quad (14)$$

where

$$\mathbf{R}_{\Theta,m}^d = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix} \quad (15)$$

is the rotary matrix with pre-defined parameters  $\Theta = \{\theta_i = 10000^{-2(i-1)/d}, i \in [1, 2, \dots, d/2]\}$ . A graphic illustration of RoPE is shown in Figure (1). Applying our RoPE to self-attention in Equation (2), we obtain:

$$\mathbf{q}_m^\top \mathbf{k}_n = (\mathbf{R}_{\Theta,m}^d \mathbf{W}_q \mathbf{x}_m)^\top (\mathbf{R}_{\Theta,n}^d \mathbf{W}_k \mathbf{x}_n) = \mathbf{x}^\top \mathbf{W}_q \mathbf{R}_{\Theta,n-m}^d \mathbf{W}_k \mathbf{x}_n \quad (16)$$

where  $\mathbf{R}_{\Theta,n-m}^d = (\mathbf{R}_{\Theta,m}^d)^\top \mathbf{R}_{\Theta,n}^d$ . Note that  $\mathbf{R}_{\Theta}^d$  is an orthogonal matrix, which ensures stability during the process of encoding position information. In addition, due to the sparsity of  $\mathbf{R}_{\Theta}^d$ , applying matrix multiplication directly as in Equation (16) is not computationally efficient; we provide another realization in theoretical explanation.

In contrast to the additive nature of position embedding method adopted in the previous works, i.e., Equations (3) to (10), our approach is multiplicative. Moreover, RoPE naturally incorporates relative position information through rotation matrix product instead of altering terms in the expanded formulation of additive position encoding when applied with self-attention.

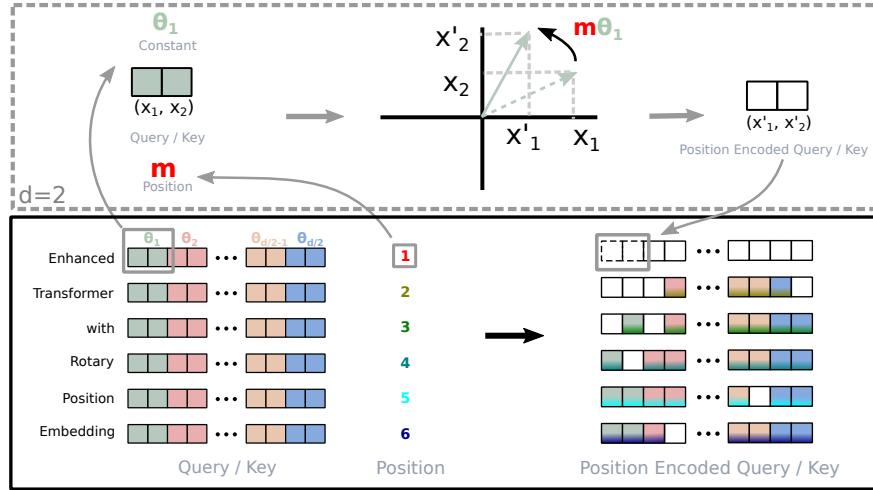


Figure 1: Implementation of Rotary Position Embedding(RoPE).

### 3.3 Properties of RoPE

**Long-term decay:** Following Vaswani et al. [2017], we set  $\theta_i = 10000^{-2i/d}$ . One can prove that this setting provides a long-term decay property (refer to Section (3.4.3) for more details), which means the inner-product will decay when the relative position increase. This property coincides with the intuition that a pair of tokens with a long relative distance should have less connection.

**RoPE with linear attention:** The self-attention can be rewritten in a more general form.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})_m = \frac{\sum_{n=1}^N \text{sim}(\mathbf{q}_m, \mathbf{k}_n) \mathbf{v}_n}{\sum_{n=1}^N \text{sim}(\mathbf{q}_m, \mathbf{k}_n)}. \quad (17)$$

The original self-attention chooses  $\text{sim}(\mathbf{q}_m, \mathbf{k}_n) = \exp(\mathbf{q}_m^\top \mathbf{k}_n / \sqrt{d})$ . Note that the original self-attention should compute the inner product of query and key for every pair of tokens, which has a quadratic complexity  $\mathcal{O}(N^2)$ . Follow Katharopoulos et al. [2020], the linear attentions reformulate Equation (17) as

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})_m = \frac{\sum_{n=1}^N \phi(\mathbf{q}_m)^\top \varphi(\mathbf{k}_n) \mathbf{v}_n}{\sum_{n=1}^N \phi(\mathbf{q}_m)^\top \varphi(\mathbf{k}_n)}, \quad (18)$$

where  $\phi(\cdot), \varphi(\cdot)$  are usually non-negative functions. The authors of Katharopoulos et al. [2020] have proposed  $\phi(x) = \varphi(x) = \text{elu}(x) + 1$  and first computed the multiplication between keys and values using the associative property of matrix multiplication. A softmax function is used in Shen et al. [2021] to normalize queries and keys separately before the inner product, which is equivalent to  $\phi(\mathbf{q}_i) = \text{softmax}(\mathbf{q}_i)$  and  $\phi(\mathbf{k}_j) = \exp(\mathbf{k}_j)$ . For more details about linear attention, we encourage readers to refer to original papers. In this section, we focus on discussing incorporating RoPE with Equation (18). Since RoPE injects position information by rotation, which keeps the norm of hidden representations unchanged, we can combine RoPE with linear attention by multiplying the rotation matrix with the outputs of the non-negative functions.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})_m = \frac{\sum_{n=1}^N (\mathbf{R}_{\Theta, m}^d \phi(\mathbf{q}_m))^\top (\mathbf{R}_{\Theta, n}^d \varphi(\mathbf{k}_n)) \mathbf{v}_n}{\sum_{n=1}^N \phi(\mathbf{q}_m)^\top \varphi(\mathbf{k}_n)}. \quad (19)$$

It is noteworthy that we keep the denominator unchanged to avoid the risk of dividing zero, and the summation in the numerator could contain negative terms. Although the weights for each value  $\mathbf{v}_i$  in Equation (19) are not strictly probabilistic normalized, we kindly argue that the computation can still model the importance of values.

### 3.4 Theoretical Explanation

#### 3.4.1 Derivation of RoPE under 2D

Under the case of  $d = 2$ , we consider two-word embedding vectors  $\mathbf{x}_q, \mathbf{x}_k$  corresponds to query and key and their position  $m$  and  $n$ , respectively. According to eq. (1), their position-encoded counterparts are:

$$\begin{aligned} \mathbf{q}_m &= f_q(\mathbf{x}_q, m), \\ \mathbf{k}_n &= f_k(\mathbf{x}_k, n), \end{aligned} \quad (20)$$

where the subscripts of  $\mathbf{q}_m$  and  $\mathbf{k}_n$  indicate the encoded positions information. Assume that there exists a function  $g$  that defines the inner product between vectors produced by  $f_{\{q, k\}}$ :

$$\mathbf{q}_m^\top \mathbf{k}_n = \langle f_q(\mathbf{x}_q, m), f_k(\mathbf{x}_k, n) \rangle = g(\mathbf{x}_m, \mathbf{x}_n, n - m), \quad (21)$$

we further require below initial condition to be satisfied:

$$\begin{aligned} \mathbf{q} &= f_q(\mathbf{x}_q, 0), \\ \mathbf{k} &= f_k(\mathbf{x}_k, 0), \end{aligned} \quad (22)$$

which can be read as the vectors with empty position information encoded. Given these settings, we attempt to find a solution of  $f_q, f_k$ . First, we take advantage of the geometric meaning of vector in 2D and its complex counter part, decompose functions in Equations (20) and (21) into:

$$\begin{aligned} f_q(\mathbf{x}_q, m) &= R_q(\mathbf{x}_q, m) e^{i\Theta_q(\mathbf{x}_q, m)}, \\ f_k(\mathbf{x}_k, n) &= R_k(\mathbf{x}_k, n) e^{i\Theta_k(\mathbf{x}_k, n)}, \\ g(\mathbf{x}_q, \mathbf{x}_k, n - m) &= R_g(\mathbf{x}_q, \mathbf{x}_k, n - m) e^{i\Theta_g(\mathbf{x}_q, \mathbf{x}_k, n - m)}, \end{aligned} \quad (23)$$

where  $R_f, R_g$  and  $\Theta_f, \Theta_g$  are the radical and angular components for  $f_{\{q, k\}}$  and  $g$ , respectively. Plug them into Equation (21), we get the relation:

$$\begin{aligned} R_q(\mathbf{x}_q, m) R_k(\mathbf{x}_k, n) &= R_g(\mathbf{x}_q, \mathbf{x}_k, n - m), \\ \Theta_k(\mathbf{x}_k, n) - \Theta_q(\mathbf{x}_q, m) &= \Theta_g(\mathbf{x}_q, \mathbf{x}_k, n - m), \end{aligned} \quad (24)$$

with the corresponding initial condition as:

$$\begin{aligned} \mathbf{q} &= \|\mathbf{q}\| e^{i\theta_q} = R_q(\mathbf{x}_q, 0) e^{i\Theta_q(\mathbf{x}_q, 0)}, \\ \mathbf{k} &= \|\mathbf{k}\| e^{i\theta_k} = R_k(\mathbf{x}_k, 0) e^{i\Theta_k(\mathbf{x}_k, 0)}, \end{aligned} \quad (25)$$

where  $\|\mathbf{q}\|$ ,  $\|\mathbf{k}\|$  and  $\theta_q, \theta_k$  are the radial and angular part of  $\mathbf{q}$  and  $\mathbf{k}$  on the 2D plane.

Next, we set  $m = n$  in Equation (24) and take into account initial conditions in Equation (25):

$$R_q(\mathbf{x}_q, m)R_k(\mathbf{x}_k, m) = R_q(\mathbf{x}_q, \mathbf{x}_k, 0) = R_k(\mathbf{x}_q, 0)R_k(\mathbf{x}_k, 0) = \|\mathbf{q}\|\|\mathbf{k}\|, \quad (26a)$$

$$\Theta_k(\mathbf{x}_k, m) - \Theta_q(\mathbf{x}_q, m) = \Theta_g(\mathbf{x}_q, \mathbf{x}_k, 0) = \|\Theta_k(\mathbf{x}_k, 0) - \Theta_q(\mathbf{x}_q, 0)\| = \|\theta_k - \theta_q\|. \quad (26b)$$

On one hand, from, a straightforward solution of  $R_f$  could be formed from Equation (26a) :

$$\begin{aligned} R_q(\mathbf{x}_q, m) &= R_q(\mathbf{x}_q, 0) = \|\mathbf{q}\| \\ R_k(\mathbf{x}_k, n) &= R_k(\mathbf{x}_k, 0) = \|\mathbf{k}\| \\ R_g(\mathbf{x}_q, \mathbf{x}_k, n - m) &= R_g(\mathbf{x}_q, \mathbf{x}_k, 0) = \|\mathbf{q}\|\|\mathbf{k}\| \end{aligned} \quad (27)$$

which interprets the radial functions  $R_q$ ,  $R_k$  and  $R_g$  are independent from the position information. On the other hand, as can be noticed in Equation (26b),  $\Theta_q(\mathbf{x}_q, m) - \theta_q = \Theta_k(\mathbf{x}_k, m) - \theta_k$  indicates that the angular functions does not dependent on query and key, we set them to  $\Theta_f := \Theta_q = \Theta_k$  and term  $\Theta_f(\mathbf{x}_{\{q,k\}}, m) - \theta_{\{q,k\}}$  is a function of position  $m$  and is independent of word embedding  $\mathbf{x}_{\{q,k\}}$ , we denote it as  $\phi(m)$ , yielding:

$$\Theta_f(\mathbf{x}_{\{q,k\}}, m) = \phi(m) + \theta_{\{q,k\}}, \quad (28)$$

Further, by plugging  $n = m + 1$  to Equation (24) and consider the above equation, we can get:

$$\phi(m + 1) - \phi(m) = \Theta_g(\mathbf{x}_q, \mathbf{x}_k, 1) + \theta_q - \theta_k, \quad (29)$$

Since RHS is a constant irrelevant to  $m$ ,  $\phi(m)$  with continuous integer inputs produce an arithmetic progression:

$$\phi(m) = m\theta + \gamma, \quad (30)$$

where  $\theta, \gamma \in \mathbb{R}$  are constants and  $\theta$  is non-zero. To summarize our solutions from Equations (27) to (30):

$$\begin{aligned} f_q(\mathbf{x}_q, m) &= \|\mathbf{q}\|e^{i\theta_q + m\theta + \gamma} = \mathbf{q}e^{i(m\theta + \gamma)}, \\ f_k(\mathbf{x}_k, n) &= \|\mathbf{k}\|e^{i\theta_k + n\theta + \gamma} = \mathbf{k}e^{i(n\theta + \gamma)}. \end{aligned} \quad (31)$$

Note that we do not apply any constrains to  $f_q$  and  $f_k$  of Equation (22), thus  $f_q(\mathbf{x}_m, 0)$  and  $f_k(\mathbf{x}_n, 0)$  are left to choose freely. To make our results comparable to Equation (3), we define:

$$\begin{aligned} \mathbf{q} &= f_q(\mathbf{x}_m, 0) = \mathbf{W}_q \mathbf{x}_m, \\ \mathbf{k} &= f_k(\mathbf{x}_n, 0) = \mathbf{W}_k \mathbf{x}_n. \end{aligned} \quad (32)$$

Then, we simply set  $\gamma = 0$  in Equation (31) of the final solution:

$$\begin{aligned} f_q(\mathbf{x}_m, m) &= (\mathbf{W}_q \mathbf{x}_m)e^{im\theta}, \\ f_k(\mathbf{x}_n, n) &= (\mathbf{W}_k \mathbf{x}_n)e^{in\theta}. \end{aligned} \quad (33)$$

### 3.4.2 Computational efficient realization of rotary matrix multiplication

Taking the advantage of the sparsity of  $\mathbf{R}_{\Theta, m}^d$  in Equation (15), a more computational efficient realization of a multiplication of  $\mathbf{R}_{\Theta}^d$  and  $\mathbf{x} \in \mathbb{R}^d$  is:

$$\mathbf{R}_{\Theta, m}^d \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix} \quad (34)$$

### 3.4.3 Long-term decay of RoPE

We can group entries of vectors  $\mathbf{q} = \mathbf{W}_q \mathbf{x}_m$  and  $\mathbf{k} = \mathbf{W}_k \mathbf{x}_n$  in pairs, and the inner product of RoPE in Equation (16) can be written as a complex number multiplication.

$$(\mathbf{R}_{\Theta, m}^d \mathbf{W}_q \mathbf{x}_m)^T (\mathbf{R}_{\Theta, n}^d \mathbf{W}_k \mathbf{x}_n) = \text{Re} \left[ \sum_{i=0}^{d/2-1} \mathbf{q}_{[2i:2i+1]} \mathbf{k}_{[2i:2i+1]}^* e^{i(m-n)\theta_i} \right] \quad (35)$$

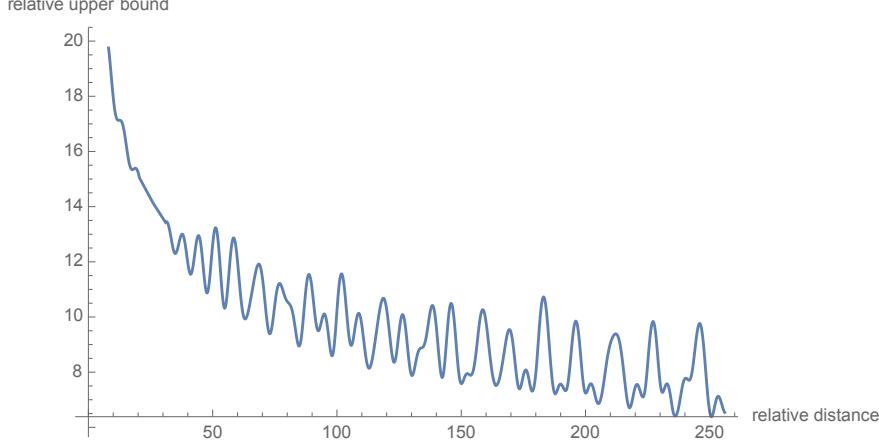


Figure 2: Long-term decay of RoPE.

where  $\mathbf{q}_{[2i:2i+1]}$  represents the  $2i^{th}$  to  $(2i+1)^{th}$  entries of  $\mathbf{q}$ . Denote  $h_i = \mathbf{q}_{[2i:2i+1]} \mathbf{k}_{[2i:2i+1]}^*$  and  $S_j = \sum_{i=0}^{j-1} e^{i(m-n)\theta_i}$ , and let  $h_{d/2} = 0$  and  $S_0 = 0$ , we can rewrite the summation using Abel transformation

$$\sum_{i=0}^{d/2-1} \mathbf{q}_{[2i:2i+1]} \mathbf{k}_{[2i:2i+1]}^* e^{i(m-n)\theta_i} = \sum_{i=0}^{d/2-1} h_i (S_{i+1} - S_i) = - \sum_{i=0}^{d/2-1} S_{i+1} (h_{i+1} - h_i). \quad (36)$$

Thus,

$$\begin{aligned} \left| \sum_{i=0}^{d/2-1} \mathbf{q}_{[2i:2i+1]} \mathbf{k}_{[2i:2i+1]}^* e^{i(m-n)\theta_i} \right| &= \left| \sum_{i=0}^{d/2-1} S_{i+1} (h_{i+1} - h_i) \right| \\ &\leq \sum_{i=0}^{d/2-1} |S_{i+1}| |(h_{i+1} - h_i)| \\ &\leq \left( \max_i |h_{i+1} - h_i| \right) \sum_{i=0}^{d/2-1} |S_{i+1}| \end{aligned} \quad (37)$$

Note that the value of  $\frac{1}{d/2} \sum_{i=1}^{d/2} |S_i|$  decay with the relative distance  $m - n$  increases by setting  $\theta_i = 10000^{-2i/d}$ , as shown in Figure (2).

## 4 Experiments and Evaluation

We evaluate the proposed RoFormer on various NLP tasks as follows. We validate the performance of the proposed solution on machine translation task Section (4.1). Then, we compare our RoPE implementation with BERTDevlin et al. [2019] during the pre-training stage in Section (4.2). Based on the pre-trained model, in Section (4.3), we further carry out evaluations across different downstream tasks from GLUE benchmarks Singh et al. [2018]. In Addition, we conduct experiments using the proposed RoPE with the linear attention of PerFormer Choromanski et al. [2020] in Section (4.4). By the end, additional tests on Chinese data are included in Section (4.5). All the experiments were run on two cloud servers with 4 x V100 GPUs.

### 4.1 Machine Translation

We first demonstrate the performance of RoFormer on sequence-to-sequence language translation tasks.

Table 1: The proposed RoFormer gives better BLEU scores compared to its baseline alternative Vaswani et al. [2017] on the WMT 2014 English-to-German translation taskBojar et al. [2014].

Model	BLEU
Transformer-baseVaswani et al. [2017]	27.3
RoFormer	<b>27.5</b>

#### 4.1.1 Experimental Settings

We choose the standard WMT 2014 English-German datasetBojar et al. [2014], which consists of approximately 4.5 million sentence pairs. We compare to the transformer-based baseline alternative Vaswani et al. [2017].

#### 4.1.2 Implementation details

We carry out some modifications on self-attention layer of the baseline model Vaswani et al. [2017] to enable RoPE to its learning process. We replicate the setup for English-to-German translation with a vocabulary of 37k based on a joint source and target byte pair encoding(BPE)Sennrich et al. [2015]. During the evaluation, a single model is obtained by averaging the last 5 checkpoints. The result uses beam search with a beam size of 4 and length penalty 0.6. We implement the experiment in PyTorch in the fairseq toolkit (MIT License)Ott et al. [2019]. Our model is optimized with the Adam optimizer using  $\beta_1 = 0.9$ ,  $\beta_2 = 0.98$ , learning rate is increased linearly from  $1e - 7$  to  $5e - 4$  and then decayed proportionally to the inverse square root of the step number. Label smoothing with 0.1 is also adopted. We report the BLEU<sup>Papineni et al. [2002]</sup> score on the test set as the final metric.

#### 4.1.3 Results

We train the baseline model and our RoFormer under the same settings and report the results in Table (1). As can be seen, our model gives better BLEU scores compared to the baseline Transformer.

### 4.2 Pre-training Language Modeling

The second experiment is to validate the performance of our proposal in terms of learning contextual representations. To achieve this, we replace the original sinusoidal position encoding of BERT with our RoPE during the pre-training step.

#### 4.2.1 Experimental Settings

We use the BookCorpus Zhu et al. [2015] and the Wikipedia Corpus Foundation [2021] from Huggingface Datasets library (Apache License 2.0) for pre-training. The corpus is further split into train and validation sets at 8:2 ratio. We use the masked language-modeling (MLM) loss values of the training process as an evaluation metric. The well-known BERT Devlin et al. [2019] is adopted as our baseline model. Note that we use bert-base-uncased in our experiments.

#### 4.2.2 Implementation details

For RoFormer, we replace the sinusoidal position encoding in the self-attention block of the baseline model with our proposed RoPE and realizes self-attention according to Equation (16). We train both BERT and RoFormer with batch size 64 and maximum sequence length of 512 for 100k steps. AdamW Loshchilov and Hutter [2017] is used as the optimizer with learning rate 1e-5.

#### 4.2.3 Results

The MLM loss during pre-training is shown on the left plot of Figure (3). Compare to the vanilla BERT, RoFormer experiences faster convergence.

### 4.3 Fine-tuning on GLUE tasks

Consistent with the previous experiments, we fine-tune the weights of our pre-trained RoFormer across various GLUE tasks in order to evaluate its generalization ability on the downstream NLP tasks.

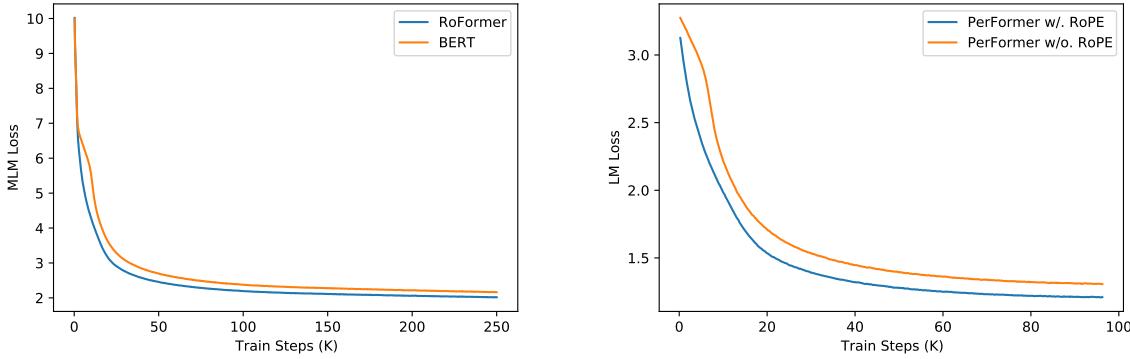


Figure 3: Evaluation of RoPE in language modeling pre-training. **Left:** training loss for BERT and RoFormer. **Right:** training loss for PerFormer with and without RoPE.

#### 4.3.1 Experimental Settings

We look at several datasets from GLUE, i.e. MRPC Dolan and Brockett [2005], SST-2 Socher et al. [2013], QNLI Rajpurkar et al. [2016], STS-B Al-Natsheh [2017], QQP Chen et al. [2018b] and MNLI Williams et al. [2018]. We use F1-score for MRPC and QQP dataset, spearman correlation for STS-B, and accuracy for the remaining as the evaluation metrics.

#### 4.3.2 Implementation details

We use Huggingface Transformers library (Apache License 2.0) Wolf et al. [2020] to fine-tune each of the aforementioned downstream tasks for 3 epochs, with a maximum sequence length of 512, batch size of 32 and learning rates 2,3,4,5e-5. Following Devlin et al. [2019], we report the best-averaged results on the validation set.

Table 2: Comparing RoFormer and BERT by fine tuning on downstream GLEU tasks.

Model	MRPC	SST-2	QNLI	STS-B	QQP	MNLI(m/mm)
BERTDevlin et al. [2019]	88.9	93.5	90.5	85.8	71.2	84.6/83.4
RoFormer	<b>89.5</b>	90.7	88.0	<b>87.0</b>	<b>86.4</b>	80.2/79.8

#### 4.3.3 Results

The evaluation results of the fine-tuning tasks are reported in Table (2). As can be seen, RoFormer can significantly outperform BERT in three out of six datasets, and the improvements are considerable.

#### 4.4 Performer with RoPE

Performer Choromanski et al. [2020] introduces an alternative attention mechanism, linear attention, which is designed to avoid quadratic computation cost that scales with input sequence length. As discussed in Section (3.3), the proposed RoPE can be easily implemented in the Performer model to realize the relative position encoding while keeping its linearly scaled complexity in self-attention. We demonstrate its performance with the pre-training task of language modeling.

#### 4.4.1 Implementation details

We carry out tests on the Enwik8 dataset Mahoney [2006], which is from English Wikipedia that includes markup, special characters and text in other languages in addition to English text. We incorporate RoPE into the 12 layer char-based Performer with 768 dimensions and 12 heads<sup>2</sup>. To better illustrate the efficacy of RoPE, we report the loss curves of the pre-training process with and without RoPE under the same settings, i.e., learning rate 1e-4, batch size 128 and a fixed maximum sequence length of 1024, etc.

<sup>2</sup>For this experiment, we adopt code (MIT License) from <https://github.com/lucidrains/performer-pytorch>

#### 4.4.2 Results

As shown on the right plot of Figure (3), substituting RoPE into Performer leads to rapid convergence and lower loss under the same amount of training steps. These improvements, in addition to the linear complexity, make Performer more attractive.

#### 4.5 Evaluation on Chinese Data

In addition to experiments on English data, we show additional results on Chinese data. To validate the performance of RoFormer on long texts, we conduct experiments on long documents whose length exceeds 512 characters.

##### 4.5.1 Implementation

In these experiments, we carried out some modifications on WoBERT Su [2020] by replacing the absolute position embedding with our proposed RoPE. As a cross-comparison with other pre-trained Transformer-based models in Chinese, i.e. BERT Devlin et al. [2019], WoBERT Su [2020], and NEZHA Wei et al. [2019], we tabulate their tokenization level and position embedding information in Table (3).

Table 3: Cross-comparison between our RoFormer and other pre-trained models on Chinese data. ‘abs’ and ‘rel’ annotates absolute position embedding and relative position embedding, respectively.

Model	BERTDevlin et al. [2019]	WoBERTSu [2020]	NEZHAWei et al. [2019]	RoFormer
Tokenization level	char	word	char	word
Position embedding	abs.	abs.	rel.	RoPE

##### 4.5.2 Pre-training

We pre-train RoFormer on approximately 34GB of data collected from Chinese Wikipedia, news and forums. The pre-training is carried out in multiple stages with changing batch size and maximum input sequence length in order to adapt the model to various scenarios. As shown in Table (4), the accuracy of RoFormer elevates with an increasing upper bound of sequence length, which demonstrates the ability of RoFormer in dealing with long texts. We claim that this is the attribute to the excellent generalizability of the proposed RoPE.

Table 4: Pre-training strategy of RoFormer on Chinese dataset. The training procedure is divided into various consecutive stages. In each stage, we train the model with a specific combination of maximum sequence length and batch size.

Stage	Max seq length	Batch size	Training steps	Loss	Accuracy
1	512	256	200k	1.73	65.0%
2	1536	256	12.5k	1.61	66.8%
3	256	256	120k	1.75	64.6%
4	128	512	80k	1.83	63.4%
5	1536	256	10k	1.58	67.4%
6	512	512	30k	1.66	66.2%

##### 4.5.3 Downstream Tasks & Dataset

We choose Chinese AI and Law 2019 Similar Case Matching (CAIL2019-SCM)Xiao et al. [2019] dataset to illustrate the ability of RoFormer in dealing with long texts, i.e., semantic text matching. CAIL2019-SCM contains 8964 triplets of cases published by the Supreme People’s Court of China. The input triplet, denoted as (A, B and C), are fact descriptions of three cases. The task is to predict whether the pair (A, B) is closer than (A, C) under a predefined similarity measure. Note that existing methods mostly cannot perform significantly on CAIL2019-SCM dataset due to the length of documents (i.e., mostly more than 512 characters). We split train, validation and test sets based on the well-known ratio 6:2:2.

##### 4.5.4 Results

We apply the pre-trained RoFormer model to CAIL2019-SCM with different input lengths. The model is compared with the pre-trained BERT and WoBERT model on the same pre-training data, as shown in Table (5). With short text cut-offs,

i.e., 512, the result from RoFormer is comparable to WoBERT and is slightly better than the BERT implementation. However, when increasing the maximum input text length to 1024, RoFormer outperforms WoBERT by an absolute improvement of 1.5%.

Table 5: Experiment results on CAIL2019-SCM task. Numbers in the first column denote the maximum cut-off sequence length. The results are presented in terms of percent accuracy.

Model	Validation	Test
BERT-512	64.13%	67.77%
WoBERT-512	64.07%	68.10%
<b>RoFormer-512</b>	64.13%	68.29%
<b>RoFormer-1024</b>	<b>66.07%</b>	<b>69.79%</b>

#### 4.5.5 Limitations of the work

Although we provide theoretical groundings as well as promising experimental justifications, our method is limited by following facts:

- Despite the fact that we mathematically format the relative position relations as rotations under 2D sub-spaces, there lacks of thorough explanations on why it converges faster than baseline models that incorporates other position encoding strategies.
- Although we have proved that our model has favourable property of long-term decay for intern-token products, Section (3.3), which is similar to the existing position encoding mechanisms, our model shows superior performance on long texts than peer models, we have not come up with a faithful explanation.

Our proposed RoFormer is built upon the Transformer-based infrastructure, which requires hardware resources for pre-training purpose.

## 5 Conclusions

In this work, we proposed a new position embedding method that incorporates explicit relative position dependency in self-attention to enhance the performance of transformer architectures. Our theoretical analysis indicates that relative position can be naturally formulated using vector production in self-attention, with absolute position information being encoded through a rotation matrix. In addition, we mathematically illustrated the advantageous properties of the proposed method when applied to the Transformer. Finally, experiments on both English and Chinese benchmark datasets demonstrate that our method encourages faster convergence in pre-training. The experimental results also show that our proposed RoFormer can achieve better performance on long texts task.

## References

- Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In *International Conference on Machine Learning*, pages 1243–1252. PMLR, 2017.
- Md. Amirul Islam, Sen Jia, and Neil D. B. Bruce. How much position information do convolutional neural networks encode? *ArXiv*, abs/2001.08248, 2020.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fdb053c1c4a845aa-Paper.pdf>.
- J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019.
- A. Radford, Jeffrey Wu, R. Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.
- Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank Reddi, and Sanjiv Kumar. Are transformers universal approximators of sequence-to-sequence functions? In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=ByxRMONTvr>.

- Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=H1eA7AEtvS>.
- Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. ELECTRA: Pre-training text encoders as discriminators rather than generators. In *ICLR*, 2020. URL <https://openreview.net/pdf?id=r1xMH1BtvB>.
- A. Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.
- Ankur P. Parikh, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit. A decomposable attention model for natural language inference. In *EMNLP*, 2016.
- Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *NAACL-HLT*, 2018.
- Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, I. Simon, C. Hawthorne, Andrew M. Dai, M. Hoffman, M. Dinucleescu, and D. Eck. Music transformer. *arXiv: Learning*, 2018.
- Zihang Dai, Z. Yang, Yiming Yang, J. Carbonell, Quoc V. Le, and R. Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *ACL*, 2019.
- Z. Yang, Zihang Dai, Yiming Yang, J. Carbonell, R. Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *NeurIPS*, 2019.
- Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, W. Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21: 140:1–140:67, 2020.
- Guolin Ke, Di He, and T. Liu. Rethinking positional encoding in language pre-training. *ArXiv*, abs/2006.15595, 2020.
- Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention. *ArXiv*, abs/2006.03654, 2020.
- Zhiheng Huang, Davis Liang, Peng Xu, and Bing Xiang. Improve transformer models with better relative position embeddings. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3327–3335, Online, November 2020. Association for Computational Linguistics. doi:10.18653/v1/2020.findings-emnlp.298. URL <https://www.aclweb.org/anthology/2020.findings-emnlp.298>.
- Xuanqing Liu, Hsiang-Fu Yu, Inderjit S. Dhillon, and Cho-Jui Hsieh. Learning to encode position for transformer with continuous dynamical model. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 6327–6335. PMLR, 2020. URL <http://proceedings.mlr.press/v119/liu20n.html>.
- Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 6572–6583, 2018a. URL <https://proceedings.neurips.cc/paper/2018/hash/69386f6bb1dfed68692a24c8686939b9-Abstract.html>.
- Benyou Wang, Donghao Zhao, Christina Lioma, Qiuchi Li, Peng Zhang, and Jakob Grue Simonsen. Encoding word order in complex embeddings. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=Hke-WTVtww>.
- Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, pages 5156–5165. PMLR, 2020.
- Zhuoran Shen, Mingyuan Zhang, Haiyu Zhao, Shuai Yi, and Hongsheng Li. Efficient attention: Attention with linear complexities. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 3531–3539, 2021.
- Amapreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. 04 2018.
- Krzysztof Choromanski, Valerii Likhoshesterov, David Dohan, Xingyou Song, A. Gane, Tamás Sarlós, Peter Hawkins, J. Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy J. Colwell, and Adrian Weller. Rethinking attention with performers. *ArXiv*, abs/2009.14794, 2020.
- Ondrej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Johannes Leveling, Christof Monz, Pavel Pecina, Matt Post, Herve Saint-Amand, Radu Soricut, Lucia Specia, and Alevs Tamchyna. Findings of the 2014 workshop on statistical machine translation. pages 12–58, 06 2014. doi:10.3115/v1/W14-3302.

- Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. 08 2015.
- Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. pages 48–53, 01 2019. doi:10.18653/v1/N19-4009.
- Kishore Papineni, Salim Roukos, Todd Ward, and Wei Jing Zhu. Bleu: a method for automatic evaluation of machine translation. 10 2002. doi:10.3115/1073083.1073135.
- Yukun Zhu, Ryan Kiros, Richard Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *arXiv preprint arXiv:1506.06724*, 2015.
- Wikimedia Foundation. Wikimedia downloads, <https://dumps.wikimedia.org>, 2021.
- Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. *arXiv e-prints*, art. arXiv:1711.05101, November 2017.
- William B. Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*, 2005. URL <https://www.aclweb.org/anthology/I05-5002>.
- Richard Socher, A. Perelygin, J.Y. Wu, J. Chuang, C.D. Manning, A.Y. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. *EMNLP*, 1631:1631–1642, 01 2013.
- Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. pages 2383–2392, 01 2016. doi:10.18653/v1/D16-1264.
- Hussein Al-Natsheh. Udl at semeval-2017 task 1: Semantic textual similarity estimation of english sentence pairs using regression model over pairwise features. 08 2017.
- Z. Chen, H. Zhang, and L. Zhang, X.and Zhao. Quora question pairs., 2018b. URL <https://www.kaggle.com/c/quora-question-pairs>.
- Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. pages 1112–1122, 01 2018. doi:10.18653/v1/N18-1101.
- Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierrick Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. URL <https://www.aclweb.org/anthology/2020.emnlp-demos.6>.
- Matt Mahoney. Large text compression benchmark, <http://www.mattmahoney.net/dc/text.html>, 2006.
- Jianlin Su. Wobert: Word-based chinese bert model - zhuiyiai. Technical report, 2020. URL <https://github.com/ZhuiyiTechnology/WoBERT>.
- Victor Junqiu Wei, Xiaozhe Ren, Xiaoguang Li, Wenyong Huang, Yi Liao, Yasheng Wang, Jiashu Lin, Xin Jiang, Xiao Chen, and Qun Liu. Nezha: Neural contextualized representation for chinese language understanding. 08 2019.
- Chaojun Xiao, Haoxi Zhong, Zhipeng Guo, Cunchao Tu, Zhiyuan Liu, Maosong Sun, Tianyang Zhang, Xianpei Han, Zhen hu, Heng Wang, and Jianfeng Xu. Cail2019-scm: A dataset of similar case matching in legal domain. 11 2019.

# SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models

Guangxuan Xiao<sup>\*1</sup> Ji Lin<sup>\*1</sup> Mickael Seznec<sup>2</sup> Hao Wu<sup>2</sup> Julien Demouth<sup>2</sup> Song Han<sup>1</sup>  
<https://github.com/mit-han-lab/smoothquant>

## Abstract

Large language models (LLMs) show excellent performance but are compute- and memory-intensive. Quantization can reduce memory and accelerate inference. However, existing methods cannot maintain accuracy and hardware efficiency at the same time. We propose SmoothQuant, a training-free, accuracy-preserving, and general-purpose post-training quantization (PTQ) solution to enable 8-bit weight, 8-bit activation (W8A8) quantization for LLMs. Based on the fact that weights are easy to quantize while activations are not, SmoothQuant smooths the activation outliers by offline *migrating* the quantization difficulty from activations to weights with a mathematically equivalent transformation. SmoothQuant enables an INT8 quantization of *both* weights and activations for all the matrix multiplications in LLMs, including OPT, BLOOM, GLM, MT-NLG, and LLaMA family. We demonstrate up to 1.56× speedup and 2× memory reduction for LLMs with negligible loss in accuracy. SmoothQuant enables serving 530B LLM within a single node. Our work offers a turn-key solution that reduces hardware costs and democratizes LLMs.

## 1 Introduction

Large-scale language models (LLMs) show excellent performance on various tasks (Brown et al., 2020a; Zhang et al., 2022). However, serving LLMs is budget and energy-consuming due to their gigantic model size. For example, the GPT-3 (Brown et al., 2020a) model contains 175B parameters, which will consume at least 350GB of memory to store and run in FP16, requiring 8×48GB A6000

<sup>\*</sup>Equal contribution <sup>1</sup>Massachusetts Institute of Technology  
<sup>2</sup>NVIDIA. Correspondence to: Guangxuan Xiao <xgx@mit.edu>, Ji Lin <jilin@mit.edu>.

*Proceedings of the 40<sup>th</sup> International Conference on Machine Learning*, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

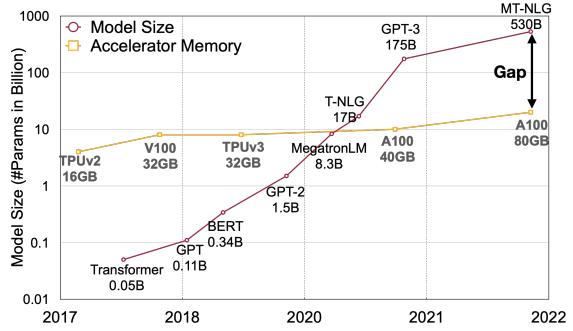


Figure 1: The model size of large language models is developing at a faster pace than the GPU memory in recent years, leading to a big gap between the supply and demand for memory. Quantization and model compression techniques can help bridge the gap.

GPUs or 5×80GB A100 GPUs just for inference. Due to the huge computation and communication overhead, the inference latency may also be unacceptable to real-world applications. *Quantization* is a promising way to reduce the cost of LLMs (Dettmers et al., 2022; Yao et al., 2022). By quantizing the *weights and activations* with low-bit integers, we can reduce GPU memory requirements, in size and bandwidth, and accelerate compute-intensive operations (i.e., GEMM in linear layers, BMM in attention). For instance, INT8 quantization of weights and activations can halve the GPU memory usage and nearly double the throughput of matrix multiplications compared to FP16.

However, unlike CNN models or smaller transformer models like BERT (Devlin et al., 2019), the *activations* of LLMs are difficult to quantize. When we scale up LLMs beyond 6.7B parameters, systematic outliers with large magnitude will emerge in activations (Dettmers et al., 2022), leading to large quantization errors and accuracy degradation. ZeroQuant (Yao et al., 2022) applies dynamic per-token activation quantization and group-wise weight quantization (defined in Figure 3 Sec. 2). It can be implemented efficiently and delivers good accuracy for GPT-3-350M and GPT-J-6B. However, it can not maintain the accuracy for the large OPT model with 175 billion parameters (see Section 5.2). LLM.int8() (Dettmers et al., 2022) addresses

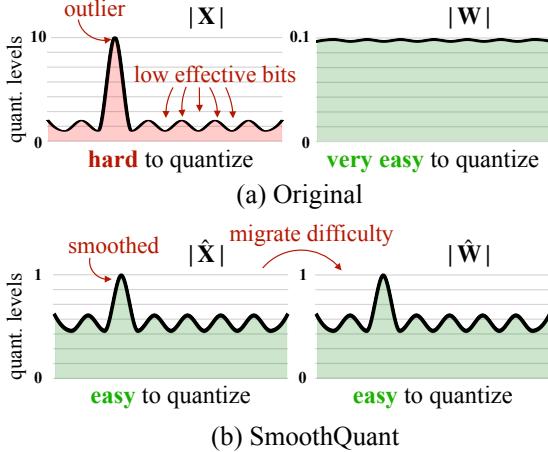


Figure 2: SmoothQuant’s intuition: the activation  $\mathbf{X}$  is hard to quantize because outliers stretch the quantization range, leaving few effective bits for most values. We migrate the scale variance from activations to weights  $\mathbf{W}$  during offline to reduce the quantization difficulty of activations. The smoothed activation  $\hat{\mathbf{X}}$  and the adjusted weight  $\hat{\mathbf{W}}$  are both easy to quantize.

that accuracy issue by further introducing a mixed-precision decomposition (i.e., it keeps outliers in FP16 and uses INT8 for the other activations). However, it is hard to implement the decomposition efficiently on hardware accelerators. Therefore, deriving an *efficient, hardware-friendly*, and preferably *training-free* quantization scheme for LLMs that would use INT8 for all the compute-intensive operations remains an open challenge.

We propose SmoothQuant, an accurate and efficient post-training quantization (PTQ) solution for LLMs. SmoothQuant relies on a key observation: even if activations are much harder to quantize than weights due to the presence of outliers (Dettmers et al., 2022), different tokens exhibit similar variations across their channels. Based on this observation, SmoothQuant offline migrates the quantization difficulty from activations to weights (Figure 2). SmoothQuant proposes a mathematically equivalent per-channel scaling transformation that significantly smooths the magnitude across the channels, making the model quantization-friendly. Since SmoothQuant is compatible with various quantization schemes, we implement three efficiency levels of quantization settings for SmoothQuant (see Table 2, O1-O3). Experiments show that SmoothQuant is hardware-efficient: it can maintain the performance of OPT-175B (Zhang et al., 2022), BLOOM-176B (Scao et al., 2022), GLM-130B (Zeng et al., 2022), and MT-NLG 530B (Smith et al., 2022), leading to up to  $1.51\times$  speed up and  $1.96\times$  memory saving on PyTorch. SmoothQuant is easy to implement. We integrate SmoothQuant into FasterTransformer, the state-of-the-art transformer serving framework, achieving up to  $1.56\times$

speedup and halving the memory usage compared with FP16. Remarkably, SmoothQuant allows serving large models like OPT-175B using only half number of GPUs compared to FP16 while being faster, and enabling the serving of a 530B model within one 8-GPU node. Our work democratizes the use of LLMs by offering a turnkey solution to reduce the serving cost. We hope SmoothQuant can inspire greater use of LLMs in the future.

## 2 Preliminaries

**Quantization** maps a high-precision value into discrete levels. We study integer uniform quantization (Jacob et al., 2018) (specifically INT8) for better hardware support and efficiency. The quantization process can be expressed as:

$$\bar{\mathbf{X}}^{\text{INT8}} = \lceil \frac{\mathbf{X}^{\text{FP16}}}{\Delta} \rceil, \quad \Delta = \frac{\max(|\mathbf{X}|)}{2^{N-1} - 1}, \quad (1)$$

where  $\mathbf{X}$  is the floating-point tensor,  $\bar{\mathbf{X}}$  is the quantized counterpart,  $\Delta$  is the quantization step size,  $\lceil \cdot \rceil$  is the rounding function, and  $N$  is the number of bits (8 in our case). Here we assume the tensor is *symmetric* at 0 for simplicity; the discussion is similar for asymmetric cases (e.g., after ReLU) by adding a zero-point (Jacob et al., 2018).

Such quantizer uses the maximum absolute value to calculate  $\Delta$  so that it preserves the outliers in activation, which are found to be important for accuracy (Dettmers et al., 2022). We can calculate  $\Delta$  offline with the activations of some calibration samples, what we call **static quantization**. We can also use the runtime statistics of activations to get  $\Delta$ , what we call **dynamic quantization**. As shown in Figure 3, quantization has different granularity levels. The **per-tensor** quantization uses a single step size for the entire matrix. We can further enable finer-grained quantization by using different quantization step sizes for activations associated with each token (**per-token** quantization) or each output channel of weights (**per-channel** quantization). A coarse-grained version of per-channel quantization is to use different quantization steps for different channel groups, called **group-wise** quantization (Shen et al., 2020; Yao et al., 2022).

For a linear layer in Transformers (Vaswani et al., 2017)  $\mathbf{Y} = \mathbf{X} \cdot \mathbf{W}$ ,  $\mathbf{Y} \in \mathbb{R}^{T \times C_o}$ ,  $\mathbf{X} \in \mathbb{R}^{T \times C_i}$ ,  $\mathbf{W} \in \mathbb{R}^{C_i \times C_o}$ , where  $T$  is the number of tokens,  $C_i$  is the input channel, and  $C_o$  is the output channel (see Figure 3, we omit the batch dimension for simplicity), we can reduce the storage by half compared to FP16 by quantizing the weights to INT8. However, to speed up the inference, we need to quantize both weights and activations into INT8 (i.e., W8A8) to utilize the integer kernels (e.g., INT8 GEMM), which are supported by a wide range of hardware (e.g., NVIDIA GPUs, Intel CPUs, Qualcomm DSPs, etc.).

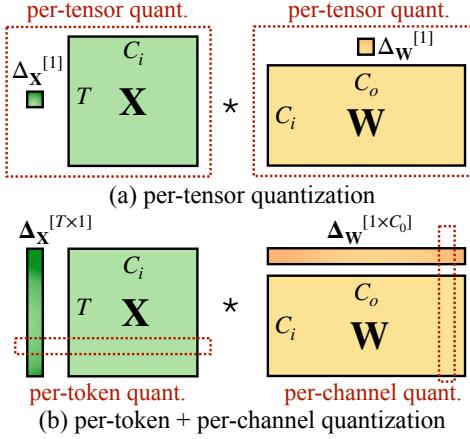


Figure 3: Definition of per-tensor, per-token, and per-channel quantization. Per-tensor quantization is the most efficient to implement. For vector-wise quantization to efficiently utilize the INT8 GEMM kernels, we can only use scaling factors from the outer dimensions (i.e., token dimension  $T$  and out channel dimension  $C_o$ ) but not inner dimension (i.e., in channel dimension  $C_i$ ).

### 3 Review of Quantization Difficulty

LLMs are notoriously difficult to quantize due to the outliers in the activations (Dettmers et al., 2022; Wei et al., 2022; Bondarenko et al., 2021). We first review the difficulties of activation quantization and look for a pattern amongst outliers. We visualize the input activations and the weights of a linear layer that has a large quantization error in Figure 4 (left). We can find several patterns that motivate our method:

**1. Activations are harder to quantize than weights.** The weight distribution is quite uniform and flat, which is easy to quantize. Previous work has shown that quantizing the weights of LLMs with INT8 or even with INT4 does not degrade accuracy (Dettmers et al., 2022; Yao et al., 2022; Zeng et al., 2022), which echoes our observation.

**2. Outliers make activation quantization difficult.** The scale of outliers in activations is  $\sim 100\times$  larger than most of the activation values. In the case of per-tensor quantization (Equation 1), the large outliers dominate the maximum magnitude measurement, leading to low *effective quantization bits/levels* (Figure 2) for non-outlier channels: suppose the maximum magnitude of channel  $i$  is  $m_i$ , and the maximum value of the whole matrix is  $m$ , the effective quantization levels of channel  $i$  is  $2^8 \cdot m_i/m$ . For non-outlier channels, the effective quantization levels would be very small (2-3), leading to large quantization errors.

**3. Outliers persist in fixed channels.** Outliers appear in a small fraction of the *channels*. If one channel has an outlier, it persistently appears in all tokens (Figure 4, red). The variance amongst the channels for a given token is large

Table 1: Among different activation quantization schemes, only per-channel quantization (Bondarenko et al., 2021) preserves the accuracy, but it is *not* compatible (marked in gray) with INT8 GEMM kernels. We report the average accuracy on WinoGrande, HellaSwag, PIQA, and LAMBADA.

Model size (OPT-)	6.7B	13B	30B	66B	175B
FP16	64.9%	65.6%	67.9%	69.5%	71.6%
INT8 per-tensor	39.9%	33.0%	32.8%	33.1%	32.3%
INT8 per-token	42.5%	33.0%	33.1%	32.9%	31.7%
INT8 per-channel	64.8%	65.6%	68.0%	69.4%	71.4%

(the activations in some channels are very large, but most are small), but the variance between the magnitudes of a given channel across tokens is small (outlier channels are consistently large). Due to the persistence of outliers and the small variance inside each channel, if we could perform *per-channel* quantization (Bondarenko et al., 2021) of the activation (i.e., using a different quantization step for each channel), the quantization error would be much smaller compared to *per-tensor* quantization, while *per-token* quantization helps little. In Table 1, we verify the assumption that *simulated* per-channel activation quantization successfully bridges the accuracy with the FP16 baseline, which echoes the findings of Bondarenko et al..

However, per-channel activation quantization does not map well to hardware-accelerated GEMM kernels, that rely on a sequence of operations executed at a high throughput (e.g., Tensor Core MMAs) and do not tolerate the insertion of instructions with a lower throughput (e.g., conversions or CUDA Core FMAs) in that sequence. In those kernels, scaling can only be performed along the outer dimensions of the matrix multiplication (i.e., token dimension of activations  $T$ , output channel dimension of weights  $C_o$ , see Figure 3), which can be applied after the matrix multiplication finishes:

$$\mathbf{Y} = \text{diag}(\Delta_{\mathbf{X}}^{\text{FP16}}) \cdot (\bar{\mathbf{X}}^{\text{INT8}} \cdot \bar{\mathbf{W}}^{\text{INT8}}) \cdot \text{diag}(\Delta_{\mathbf{W}}^{\text{FP16}}) \quad (2)$$

Therefore, previous works all use per-token activation quantization for linear layers (Dettmers et al., 2022; Yao et al., 2022), although they cannot address the difficulty of activation quantization (only slightly better than per-tensor).

### 4 SmoothQuant

Instead of per-channel activation quantization (which is infeasible), we propose to “smooth” the input activation by dividing it by a per-channel smoothing factor  $\mathbf{s} \in \mathbb{R}^{C_i}$ . To keep the mathematical equivalence of a linear layer, we scale the weights accordingly in the reversed direction:

$$\mathbf{Y} = (\mathbf{X} \text{diag}(\mathbf{s})^{-1}) \cdot (\text{diag}(\mathbf{s}) \mathbf{W}) = \hat{\mathbf{X}} \hat{\mathbf{W}} \quad (3)$$

Considering input  $\mathbf{X}$  is usually produced from previous linear operations (e.g., linear layers, layer norms, etc.), we

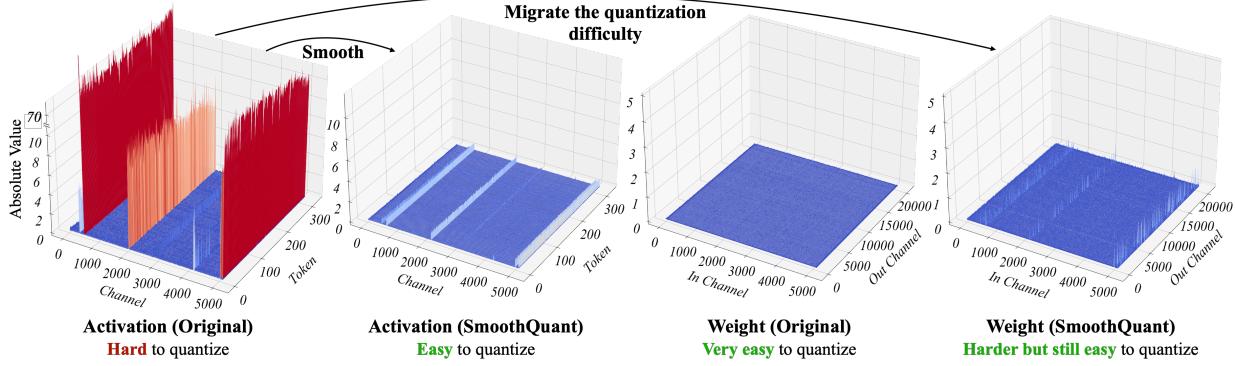


Figure 4: Magnitude of the input activations and weights of a linear layer in OPT-13B before and after SmoothQuant. Observations: (1) there are a few channels in the original activation map whose magnitudes are very large (greater than 70); (2) the variance in one activation channel is small; (3) the original weight distribution is flat and uniform. SmoothQuant migrates the outlier channels from activation to weight. In the end, the outliers in the activation are greatly smoothed while the weight is still pretty smooth and flat.

can easily fuse the smoothing factor into previous layers’ parameters *offline*, which does not incur kernel call overhead from an extra scaling. For some other cases, when the input is from a residual add, we can add an extra scaling to the residual branch similar to Wei et al. (2022).

**Migrate the quantization difficulty from activations to weights.** We aim to choose a per-channel smoothing factor  $s$  such that  $\hat{\mathbf{X}} = \mathbf{X}\text{diag}(\mathbf{s})^{-1}$  is easy to quantize. To reduce the quantization error, we should *increase the effective quantization bits* for all the channels. The total effective quantization bits would be largest when all the channels have the same maximum magnitude. Therefore, a straight-forward choice is  $s_j = \max(|\mathbf{X}_j|)$ ,  $j = 1, 2, \dots, C_i$ , where  $j$  corresponds to  $j$ -th input channel. This choice ensures that after the division, all the activation channels will have the same maximum value, which is easy to quantize. Note that the range of activations is dynamic; it varies for different input samples. Here, we estimate the scale of activations channels using calibration samples from the pre-training dataset (Jacob et al., 2018). However, this formula pushes *all* the quantization difficulties to the weights. We find that, in this case, the quantization errors would be large for the weights (outlier channels are migrated to weights now), leading to a large accuracy degradation (see Figure 10). On the other hand, we can also push all the quantization difficulty from weights to activations by choosing  $s_j = 1/\max(|\mathbf{W}_j|)$ . Similarly, the model performance is bad due to the activation quantization errors. Therefore, we need to *split* the quantization difficulty between weights and activations so that they are both easy to quantize.

Here we introduce a hyper-parameter, migration strength  $\alpha$ , to control how much difficulty we want to migrate from activation to weights, using the following equation:

$$s_j = \max(|\mathbf{X}_j|)^\alpha / \max(|\mathbf{W}_j|)^{1-\alpha} \quad (4)$$

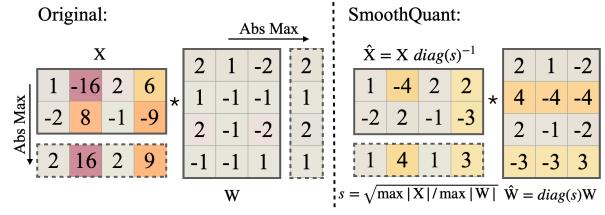


Figure 5: Main idea of SmoothQuant when  $\alpha$  is 0.5. The smoothing factor  $s$  is obtained on calibration samples and the entire transformation is performed offline. At runtime, the activations are smooth without scaling.

We find that for most of the models, e.g., all OPT (Zhang et al., 2022) and BLOOM (Scao et al., 2022) models,  $\alpha = 0.5$  is a well-balanced point to evenly split the quantization difficulty, especially when we are using the same quantizer for weights and activations (e.g., per-tensor, static quantization). The formula ensures that the weights and activations at the corresponding channel share a similar maximum value, thus sharing the same quantization difficulty. Figure 5 illustrates the smoothing transformation when we take  $\alpha = 0.5$ . For some other models where activation outliers are more significant (e.g., GLM-130B (Zeng et al., 2022) has  $\sim 30\%$  outliers, which are more difficult for activation quantization), we can choose a larger  $\alpha$  to migrate more quantization difficulty to weights (like 0.75).

**Applying SmoothQuant to Transformer blocks.** Linear layers take up most of the parameters and computation of LLM models. By default, we perform scale smoothing for the input activations of self-attention and feed-forward layers and quantize all linear layers with W8A8. We also quantize BMM operators in the attention computation. We design a quantization flow for transformer blocks in Figure 6. We quantize the inputs and weights of compute-heavy operators like linear layers and BMM in attention layers with INT8,

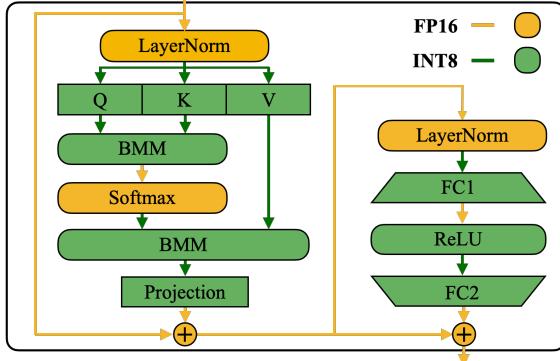


Figure 6: SmoothQuant’s precision mapping for a Transformer block. All compute-intensive operators like linear layers and batched matmul (BMMs) use INT8 arithmetic.

Table 2: Quantization setting of the baselines and SmoothQuant. All weight and activations use INT8 representations unless specified. For SmoothQuant, the efficiency **improves** from O1 to O3 (i.e., lower latency).

Method	Weight	Activation
W8A8	per-tensor	per-tensor dynamic
ZeroQuant	group-wise	per-token dynamic
LLM.int8()	per-channel	per-token dynamic+FP16
Outlier Suppression	per-tensor	per-tensor static
SmoothQuant-O1	per-tensor	per-token dynamic
SmoothQuant-O2	per-tensor	per-tensor dynamic
SmoothQuant-O3	per-tensor	per-tensor static

while keeping the activation as FP16 for other lightweight element-wise operations like ReLU, Softmax, and Layer-Norm. Such a design helps us to balance accuracy and inference efficiency.

## 5 Experiments

### 5.1 Setups

**Baselines.** We compare with four baselines in the INT8 post-training quantization setting, i.e., without re-training of the model parameters: W8A8 naive quantization, ZeroQuant (Yao et al., 2022), LLM.int8 () (Dettmers et al., 2022), and Outlier Suppression (Wei et al., 2022). Since SmoothQuant is orthogonal to the quantization schemes, we provide gradually aggressive and efficient quantization levels from O1 to O3. The detailed quantization schemes of the baselines and SmoothQuant are shown in Table 2.

**Models and datasets.** We choose three families of LLMs to evaluate SmoothQuant: OPT (Zhang et al., 2022), BLOOM (Scao et al., 2022), and GLM-130B (Zeng et al., 2022). We use seven zero-shot evaluation tasks: LAMBADA (Paperno et al., 2016), HellaSwag (Zellers

et al., 2019), PIQA (Bisk et al., 2020), WinoGrande (Sakaguchi et al., 2019), OpenBookQA (Mihaylov et al., 2018), RTE (Wang et al., 2018), COPA (Roemmele et al., 2011), and one language modeling dataset WikiText (Merity et al., 2016) to evaluate the OPT and BLOOM models. We use MMLU (Hendrycks et al., 2020), MNLI (Williams et al., 2018), QNLI (Wang et al., 2018) and LAMBADA to evaluate the GLM-130B model because some of the aforementioned benchmarks appear in the training set of GLM-130B. We use lm-eval-harness\* to evaluate OPT and BLOOM models, and GLM-130B’s official repo† for its own evaluation. Finally, we scale up our method to MT-NLG 530B (Smith et al., 2022) and for the first time enabling the serving of a >500B model within a single node. Note that we focus on the *relative* performance change before and after quantization but not the absolute value.

**Activation smoothing.** The migration strength  $\alpha = 0.5$  is a general sweet spot for all the OPT and BLOOM models, and  $\alpha = 0.75$  for GLM-130B since its activations are more difficult to quantize (Zeng et al., 2022). We get a suitable  $\alpha$  by running a quick grid search on a subset of the Pile (Gao et al., 2020) validation set. To get the statistics of activations, we calibrate the smoothing factors and the static quantization step sizes *once* with 512 random sentences from the pre-training dataset Pile, and apply the same smoothed and quantized model for all downstream tasks. In this way, we can benchmark the generality and zero-shot performance of the quantized LLMs.

**Implementation.** We implement SmoothQuant with two backends: (1) PyTorch Huggingface‡ for the proof of concept, and (2) FasterTransformer§, as an example of a high-performance framework used in production environments. In both PyTorch Huggingface and FasterTransformer frameworks, we implement INT8 linear modules and the batched matrix multiplication (BMM) function with CUTLASS INT8 GEMM kernels. We simply replace the original floating point (FP16) linear modules and the `bmm` function with our INT8 kernels as the INT8 model.

### 5.2 Accurate Quantization

**Results of OPT-175B.** SmoothQuant can handle the quantization of very large LLMs, whose activations are more difficult to quantize. We study quantization on OPT-175B. As shown in Table 3, SmoothQuant can match the FP16 accuracy on all evaluation datasets with all quantization schemes. LLM.int8 () can match the floating point accuracy because they use floating-point values to represent outliers, which leads to a large latency overhead (Table 10).

\*<https://github.com/EleutherAI/lm-evaluation-harness>

†<https://github.com/THUDM/GLM-130B>

‡<https://github.com/huggingface/transformers>

§<https://github.com/NVIDIA/FasterTransformer>

Table 3: SmoothQuant maintains the accuracy of OPT-175B model after INT8 quantization, even with the most aggressive and most efficient O3 setting (Table 2). We extensively benchmark the performance on 7 zero-shot benchmarks (by reporting the average accuracy) and 1 language modeling benchmark (perplexity). \*For ZeroQuant, we also tried leaving the input activation of self-attention in FP16 and quantizing the rest to INT8, which is their solution to the GPT-NeoX-20B. But this does not solve the accuracy degradation of OPT-175B.

<i>OPT-175B</i>	LAMBADA	HellaSwag	PIQA	WinoGrande	OpenBookQA	RTE	COPA	Average↑	WikiText↓
FP16	74.7%	59.3%	79.7%	72.6%	34.0%	59.9%	88.0%	66.9%	10.99
W8A8	0.0%	25.6%	53.4%	50.3%	14.0%	49.5%	56.0%	35.5%	93080
ZeroQuant	0.0%*	26.0%	51.7%	49.3%	17.8%	50.9%	55.0%	35.8%	84648
LLM.int8()	74.7%	59.2%	79.7%	72.1%	34.2%	60.3%	87.0%	66.7%	11.10
Outlier Suppression	0.00%	25.8%	52.5%	48.6%	16.6%	53.4%	55.0%	36.0%	96151
SmoothQuant-O1	74.7%	59.2%	79.7%	71.2%	33.4%	58.1%	89.0%	66.5%	11.11
SmoothQuant-O2	75.0%	59.0%	79.2%	71.2%	33.0%	59.6%	88.0%	66.4%	11.14
SmoothQuant-O3	74.6%	58.9%	79.7%	71.2%	33.4%	59.9%	90.0%	66.8%	11.17

Table 4: SmoothQuant works for different LLMs. We can quantize the 3 largest, openly available LLM models into INT8 without degrading the accuracy. For OPT-175B and BLOOM-176B, we show the average accuracy on WinoGrande, HellaSwag, PIQA, and LAMBADA. For GLM-130B we show the average accuracy on LAMBADA, MMLU, MNLI, and QNLI. \*Accuracy is not column-wise comparable due to different datasets.

Method	OPT-175B	BLOOM-176B	GLM-130B*
FP16	71.6%	68.2%	73.8%
W8A8	32.3%	64.2%	26.9%
ZeroQuant	31.7%	67.4%	26.7%
LLM.int8()	71.4%	68.0%	73.8%
Outlier Suppression	31.7%	54.1%	63.5%
SmoothQuant-O1	<b>71.2%</b>	68.3%	<b>73.7%</b>
SmoothQuant-O2	71.1%	<b>68.4%</b>	72.5%
SmoothQuant-O3	71.1%	67.4%	72.8%

The W8A8, ZeroQuant, and Outlier Suppression baselines produce nearly random results, indicating that naively quantizing the activation of LLMs will destroy the performance.

**Results of different LLMs.** SmoothQuant can be applied to various LLM designs. In Table 4, we show SmoothQuant can quantize all existing open LLMs beyond 100B parameters. Compared with the OPT-175B model, the BLOOM-176B model is easier to quantize: none of the baselines completely destroys the model; even the naive W8A8 per-tensor dynamic quantization only degrades the accuracy by 4%. The O1 and O2 levels of SmoothQuant successfully maintain the floating point accuracy, while the O3 level (per-tensor static) degrades the average accuracy by 0.8%, which we attribute to the discrepancy between the statically collected statistics and the real evaluation samples’ activation statistics. On the contrary, the GLM-130B model is more difficult to quantize (which echos Zeng et al.). Nonethe-

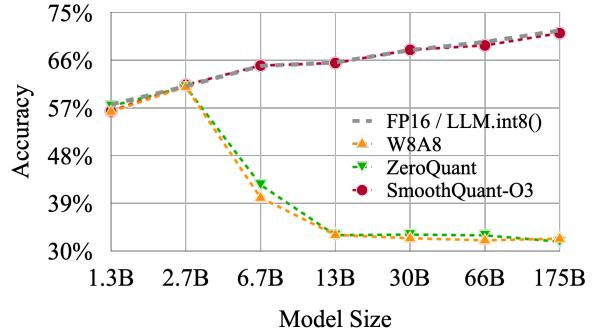


Figure 7: SmoothQuant-O3 (the most efficient setting, defined in Table 2) preserves the accuracy of OPT models across different scales when quantized to INT8. LLM.int8() requires mixed precision and suffers from slowing down.

less, SmoothQuant-O1 can match the FP16 accuracy, while SmoothQuant-O3 only degrades the accuracy by 1%, which significantly outperforms the baselines. Note that we clip the top 2% tokens when calibrating the static quantization step sizes for GLM-130B following Wei et al. (2022). Note that different model/training designs have different quantization difficulties, which we hope will inspire future research.

**Results on LLMs of different sizes.** SmoothQuant works not only for the very large LLMs beyond 100B parameters, but it also works consistently for smaller LLMs. In Figure 7, we show that SmoothQuant can work on all scales of OPT models, matching the FP16 accuracy with INT8 quantization.

**Results on Instruction-Tuned LLM** Shown in Table 5, SmoothQuant also works on instruction-tuned LLMs. We test SmoothQuant on the OPT-IML-30B model using the WikiText-2 and LAMBADA datasets. Our results show that SmoothQuant successfully preserves model accuracy

Table 5: SmoothQuant’s performance on the OPT-IML model.

OPT-IML-30B	LAMBADA $\uparrow$	WikiText $\downarrow$
FP16	69.12%	14.26
W8A8	4.21%	576.53
ZeroQuant	5.12%	455.12
LLM.int8()	69.14%	14.27
Outlier Suppression	0.00%	9485.62
SmoothQuant-O3	<b>69.77%</b>	<b>14.37</b>

Table 6: SmoothQuant can enable lossless W8A8 quantization for LLaMA models (Touvron et al., 2023). Results are perplexity on WikiText-2 dataset. We used per-token activation quantization and  $\alpha=0.8$  for SmoothQuant.

Wiki PPL $\downarrow$	7B	13B	30B	65B
FP16	11.51	10.05	7.53	6.17
W8A8 SmoothQuant	11.56	10.08	7.56	6.20

with W8A8 quantization, whereas the baselines fail to do so. SmoothQuant is a general method designed to balance the quantization difficulty for Transformer models. As the architecture of instruction-tuned LLMs is not fundamentally different from vanilla LLMs, and their pre-training processes are very similar, SmoothQuant is applicable to instruction-tuned LLMs as well.

**Results on LLaMA models.** LLaMA models are new open language models with superior performance (Touvron et al., 2023). Through initial experiments, we find LLaMA models generally have less severe activation outlier issues compared to models like OPT and BLOOM. Nonetheless, SmoothQuant still works quite well for LLaMA models. We provide some initial results of LLaMA W8A8 quantization in Table 6. SmoothQuant enables W8A8 quantization at a negligible performance degradation.

### 5.3 Speedup and Memory Saving

In this section, we show the measured speedup and memory saving of SmoothQuant-O3 integrated into PyTorch and FasterTransformer.

**Context-stage: PyTorch Implementation.** We measure the end-to-end latency of generating all hidden states for a batch of 4 sentences in one pass, i.e., the context stage latency. We record the (aggregated) peak GPU memory usage in this process. We only compare SmoothQuant with `LLM.int8()` because it is the only existing quantization method that can preserve LLM accuracy at all scales. Due to the lack of support for model parallelism in Huggingface, we only measure SmoothQuant’s performance on a single GPU for the PyTorch implementation, so we choose

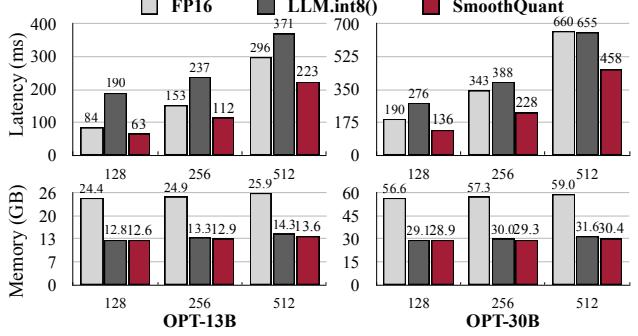


Figure 8: The PyTorch implementation of SmoothQuant-O3 achieves up to  $1.51\times$  speedup and  $1.96\times$  memory saving for OPT models on a single NVIDIA A100-80GB GPU, while `LLM.int8()` slows down the inference in most cases.

OPT-6.7B, OPT-13B, and OPT-30B for evaluation. In the FasterTransformer library, SmoothQuant can seamlessly work with Tensor Parallelism (Shoeybi et al., 2019) algorithm, so we test SmoothQuant on OPT-13B, OPT-30B, OPT-66B, and OPT-175B for both single and multi-GPU benchmarks. All our experiments are conducted on NVIDIA A100 80GB GPU servers.

In Figure 8, we show the inference latency and peak memory usage based on the PyTorch implementation. SmoothQuant is consistently faster than the FP16 baseline, getting a 1.51x speedup on OPT-30B when the sequence length is 256. We also see a trend that the larger the model, the more significant the acceleration. On the other hand, `LLM.int8()` is almost always slower than the FP16 baseline, which is due to the large overhead of the mixed-precision activation representation. In terms of memory, SmoothQuant and `LLM.int8()` can all nearly halve the memory usage of the FP16 model, while SmoothQuant saves slightly more memory because it uses fully INT8 GEMMs.

**Context-stage: FasterTransformer Implementation.** As shown in Figure 9 (top), compared to FasterTransformer’s FP16 implementation of OPT, SmoothQuant-O3 can further reduce the execution latency of OPT-13B and OPT-30B by up to  $1.56\times$  when using a single GPU. This is challenging since FasterTransformer is already more than 3x faster compared to the PyTorch implementation for OPT-30B. Remarkably, for bigger models that have to be distributed across multiple GPUs, SmoothQuant achieves similar or even better latency using only *half* the number of GPUs (1 GPU instead of 2 for OPT-66B, 4 GPUs instead of 8 for OPT-175B). This could greatly lower the cost of serving LLMs. The amount of memory needed when using SmoothQuant-O3 in FasterTransformer is reduced by a factor of almost 2x, as shown on Figure 9 (bottom).

**Decoding-stage.** In Table 7, we show SmoothQuant can significantly accelerate the autoregressive decoding stage

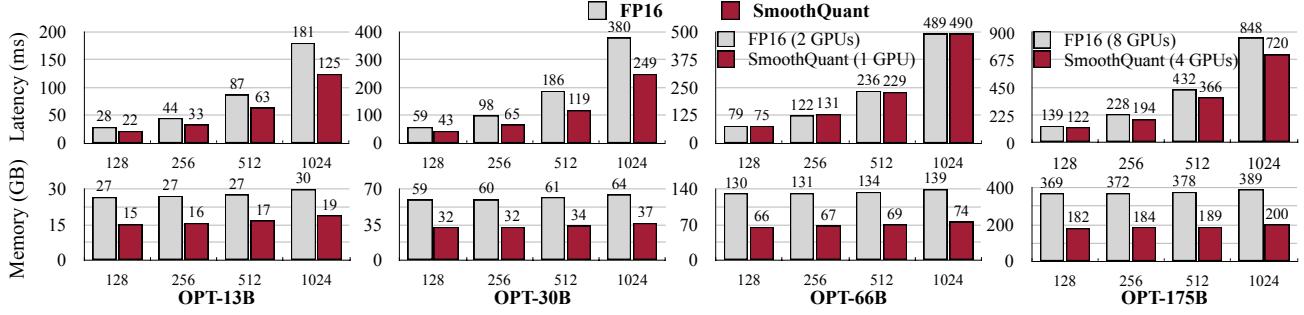


Figure 9: Inference latency (top) and memory usage (bottom) of the FasterTransformer implementation on NVIDIA A100-80GB GPUs. For smaller models, the latency can be significantly reduced with SmoothQuant-O3 by up to 1.56x compared to FP16. For the bigger models (OPT-66B and 175B), we can achieve similar or even faster inference using only **half** number of GPUs. Memory footprint is almost halved compared to FP16.

Table 7: SmoothQuant’s performance in the decoding stage.

BS	SeqLen	Latency (ms)		Memory (GB)			
		FP16	Ours	Speedup (↑)	FP16	Ours	
OPT-30B (1 GPU)							
1	512	422	314	1.35×	57	30	1.91×
1	1024	559	440	1.27×	58	31	1.87×
16	512	2488	1753	1.42×	69	44	1.59×
16	1024	OOM	3947	-	OOM	61	-
OPT-175B (8 GPUs)							
1	512	426	359	1.19×	44	23	1.87×
1	1024	571	475	1.20×	44	24	1.85×
16	512	2212	1628	1.36×	50	30	1.67×
16	1024	4133	3231	1.28×	56	37	1.52×

of LLMs. SmoothQuant constantly reduces the per-token decoding latency compared to FP16 (up to 1.42x speedup). Additionally, SmoothQuant halves the memory footprints for LLM inference, enabling the deployment of LLMs at a significantly lower cost.

Table 8: SmoothQuant can quantize MT-NLG 530B to W8A8 with negligible accuracy loss.

	LAMBADA	HellaSwag	PIQA	WinoGrande	Average
FP16	76.6%	62.1%	81.0%	72.9%	73.1%
INT8	77.2%	60.4%	80.7%	74.1%	73.1%

#### 5.4 Scaling Up: 530B Model Within a Single Node

We can further scale up SmoothQuant beyond 500B-level models, enabling efficient and accurate W8A8 quantization of MT-NLG 530B (Smith et al., 2022). As shown in Table 8 and 9, SmoothQuant enables W8A8 quantization of the 530B model at a negligible accuracy loss. The reduced model size allows us to serve the model using half number of the GPUs (16 to 8) at a similar latency, enabling the serving of a >500B model within a single node ( $8 \times$ A100 80GB GPUs).

Table 9: When serving MT-NLG 530B, SmoothQuant can reduce the memory by half at a similar latency using **half** number of GPUs, which allows serving the 530B model within a single node.

SeqLen	Prec.	#GPUs	Latency	Memory
128	FP16	16	232ms	1040GB
	INT8	8	253ms	527GB
256	FP16	16	451ms	1054GB
	INT8	8	434ms	533GB
512	FP16	16	838ms	1068GB
	INT8	8	839ms	545GB
1024	FP16	16	1707ms	1095GB
	INT8	8	1689ms	570GB

#### 5.5 Ablation Study

**Quantization schemes.** Table 10 shows the inference latency of different quantization schemes based on our PyTorch implementation. We can see that the coarser the quantization granularity (from O1 to O3), the lower the latency. And static quantization can significantly accelerate inference compared with dynamic quantization because we no longer need to calculate the quantization step sizes at runtime. SmoothQuant is faster than FP16 baseline under all settings, while `LLM.int8()` is usually slower. We recommend using a coarser scheme if the accuracy permits.

**Migration strength.** We need to find a suitable migration strength  $\alpha$  (see Equation 4) to balance the quantization difficulty of weights and activations. We ablate the effect of different  $\alpha$ ’s on OPT-175B with LAMBADA in Figure 10. When  $\alpha$  is too small (<0.4), the activations are hard to quantize; when  $\alpha$  is too large (>0.6), the weights will be hard to quantize. Only when we choose  $\alpha$  from the sweet spot region (0.4-0.6) can we get small quantization errors for both weights and activations, and maintain the model performance after quantization.

Table 10: GPU Latency (ms) of different quantization schemes. The coarser the quantization scheme (from per-token to per-tensor, dynamic to static, O1 to O3, defined in Table 2), the lower the latency. SmoothQuant achieves lower latency compared to FP16 under all settings, while `LLM.int8()` is mostly slower. The batch size is 4.

Model	OPT-13B		OPT-30B	
	256	512	256	512
FP16	152.6	296.3	343.0	659.9
<code>LLM.int8()</code>	237.1	371.5	387.9	654.9
SmoothQuant-O1	124.5	243.3	246.7	490.7
SmoothQuant-O2	120.5	235.1	240.2	478.3
SmoothQuant-O3	112.1	223.1	227.6	458.4

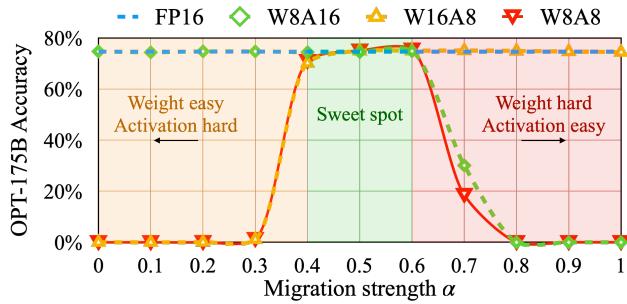


Figure 10: A suitable migration strength  $\alpha$  (sweet spot) makes both activations and weights easy to quantize. If the  $\alpha$  is too large, weights will be hard to quantize; if too small, activations will be hard to quantize.

## 6 Related Work

**Large language models (LLMs).** Pre-trained language models have achieved remarkable performance on various benchmarks by *scaling up*. GPT-3 (Brown et al., 2020b) is the first LLM beyond 100B parameters and achieves impressive few-shot/zero-shot learning results. Later works (Rae et al., 2021; Smith et al., 2022; Du et al., 2022; Chowdhery et al., 2022) continue to push the frontier of scaling, going beyond 500B parameters. However, as the language model gets larger, serving such models for inference becomes expensive and challenging. In this work, we show that our proposed method can quantize the three largest, openly available LLMs: OPT-175B (Zhang et al., 2022), BLOOM-176B (Scao et al., 2022) and GLM-130B (Zeng et al., 2022), and even MT-NLG 530B (Smith et al., 2022) to reduce the memory cost and accelerate inference.

**Model quantization.** Quantization is an effective method for reducing the model size and accelerating inference. It proves to be effective for various convolutional neural networks (CNNs) (Han et al., 2016; Jacob et al., 2018; Nagel et al., 2019; Wang et al., 2019; Lin et al., 2020) and transformers (Shen et al., 2020; Kim et al., 2021; Liu et al., 2021; Wang et al., 2020; Bondarenko et al., 2021). Weight equal-

ization (Nagel et al., 2019) and channel splitting (Zhao et al., 2019) reduce quantization error by suppressing the outliers in weights. However, these techniques cannot address the activation outliers, which are the major quantization bottleneck for LLMs (Dettmers et al., 2022).

**Quantization of LLMs.** GPTQ (Frantar et al., 2022) applies quantization only to weights but not activations (please find a short discussion in Appendix A). Zero-Quant (Yao et al., 2022) and nuQmm (Park et al., 2022) use a per-token and group-wise quantization scheme for LLMs, which requires customized CUDA kernels. Their largest evaluated models are 20B and 2.7B, respectively and fail to maintain the performance of LLMs like OPT-175B. `LLM.int8()` (Dettmers et al., 2022) uses mixed INT8/FP16 decomposition to address the activation outliers. However, such implementation leads to large latency overhead, which can be even slower than FP16 inference. Outlier Suppression (Wei et al., 2022) uses the non-scaling Layer-Norm and token-wise clipping to deal with the activation outliers. However, it only succeeds on small language models such as BERT (Devlin et al., 2019) and BART (Lewis et al., 2019) and fails to maintain the accuracy for LLMs (Table 4). Our algorithm preserves the performance of LLMs (up to 176B, the largest open-source LLM we can find) with an efficient per-tensor, static quantization scheme without retraining, allowing us to use off-the-shelf INT8 GEMM to achieve high hardware efficiency.

## 7 Conclusion

We propose SmoothQuant, an accurate and efficient post-training quantization method to enable lossless 8-bit weight and activation quantization for LLMs up to 530B parameters. SmoothQuant enables the quantization for both weight and activations for all GEMMs in the LLMs, which significantly reduces the inference latency and memory usage compared with the mixed-precision activation quantization baseline. We integrate SmoothQuant into PyTorch and FasterTransformer, getting up to  $1.56\times$  inference acceleration and halving the memory footprint. SmoothQuant democratizes the application of LLMs by offering a turnkey solution to reduce the serving cost.

## Acknowledgements

We thank MIT-IBM Watson AI Lab, MIT AI Hardware Program, Amazon and MIT Science Hub, NVIDIA Academic Partnership Award, Qualcomm Innovation Fellowship, Microsoft Turing Academic Program, and NSF for supporting this research. We thank Haotian Tang, Aohan Zeng, Eric Lin and Jilei Hou for the helpful discussions.

## References

- Bisk, Y., Zellers, R., Bras, R. L., Gao, J., and Choi, Y. Piqa: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.
- Bondarenko, Y., Nagel, M., and Blankevoort, T. Understanding and overcoming the challenges of efficient transformer quantization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 7947–7969, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. URL <https://aclanthology.org/2021.emnlp-main.627>.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020a. URL <https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf>.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020b.
- Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.
- Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.
- Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT 2019*, pp. 4171–4186. Association for Computational Linguistics, 2019.
- Du, N., Huang, Y., Dai, A. M., Tong, S., Lepikhin, D., Xu, Y., Krikun, M., Zhou, Y., Yu, A. W., Firat, O., et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, pp. 5547–5569. PMLR, 2022.
- Frantar, E., Ashkboos, S., Hoefer, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.
- Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.
- Han, S., Mao, H., and Dally, W. J. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *ICLR*, 2016.
- Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., and Steinhardt, J. Measuring massive multitask language understanding. *CoRR*, abs/2009.03300, 2020. URL <https://arxiv.org/abs/2009.03300>.
- Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018.
- Kim, S., Gholami, A., Yao, Z., Mahoney, M. W., and Keutzer, K. I-bert: Integer-only bert quantization. In *International conference on machine learning*, pp. 5506–5518. PMLR, 2021.
- Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.
- Lin, J., Chen, W.-M., Lin, Y., Gan, C., Han, S., et al. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems*, 33:11711–11722, 2020.
- Liu, Z., Wang, Y., Han, K., Zhang, W., Ma, S., and Gao, W. Post-training quantization for vision transformer. *Advances in Neural Information Processing Systems*, 34: 28092–28103, 2021.
- Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models, 2016.
- Mihaylov, T., Clark, P., Khot, T., and Sabharwal, A. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *EMNLP*, 2018.
- Nagel, M., Baalen, M. v., Blankevoort, T., and Welling, M. Data-free quantization through weight equalization and bias correction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 1325–1334, 2019.

- Paperno, D., Kruszewski, G., Lazaridou, A., Pham, N. Q., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., and Fernández, R. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1525–1534, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1144. URL <https://aclanthology.org/P16-1144>.
- Park, G., Park, B., Kwon, S. J., Kim, B., Lee, Y., and Lee, D. nuqmm: Quantized matmul for efficient inference of large-scale generative language models. *arXiv preprint arXiv:2206.09557*, 2022.
- Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*, 2022.
- Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring, R., Young, S., et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.
- Roemmele, M., Bejan, C. A., and Gordon, A. S. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *Logical Formalizations of Commonsense Reasoning, Papers from the 2011 AAAI Spring Symposium, Technical Report SS-11-06, Stanford, California, USA, March 21-23, 2011*. AAAI, 2011. URL <http://www.aaai.org/ocs/index.php/SSS/SSS11/paper/view/2418>.
- Sakaguchi, K., Bras, R. L., Bhagavatula, C., and Choi, Y. Winogrande: An adversarial winograd schema challenge at scale. *arXiv preprint arXiv:1907.10641*, 2019.
- Scao, T. L., Fan, A., Akiki, C., Pavlick, E., Ilić, S., Hesslow, D., Castagné, R., Luccioni, A. S., Yvon, F., Gallé, M., et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.
- Shen, S., Dong, Z., Ye, J., Ma, L., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 8815–8821, 2020.
- Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. URL <http://arxiv.org/abs/1909.08053>.
- Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.
- Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *CoRR*, abs/1804.07461, 2018. URL <http://arxiv.org/abs/1804.07461>.
- Wang, H., Zhang, Z., and Han, S. Spatten: Efficient sparse attention architecture with cascade token and head pruning. *CoRR*, abs/2012.09852, 2020. URL <https://arxiv.org/abs/2012.09852>.
- Wang, K., Liu, Z., Lin, Y., Lin, J., and Han, S. HAQ: Hardware-Aware Automated Quantization with Mixed Precision. In *CVPR*, 2019.
- Wei, X., Zhang, Y., Zhang, X., Gong, R., Zhang, S., Zhang, Q., Yu, F., and Liu, X. Outlier suppression: Pushing the limit of low-bit transformer language models, 2022. URL <https://arxiv.org/abs/2209.13325>.
- Williams, A., Nangia, N., and Bowman, S. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 1112–1122. Association for Computational Linguistics, 2018. URL <http://aclweb.org/anthology/N18-1101>.
- Yao, Z., Aminabadi, R. Y., Zhang, M., Wu, X., Li, C., and He, Y. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers, 2022. URL <https://arxiv.org/abs/2206.01861>.
- Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.
- Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., and Choi, Y. Hellaswag: Can a machine really finish your sentence? *CoRR*, abs/1905.07830, 2019. URL <http://arxiv.org/abs/1905.07830>.

Zeng, A., Liu, X., Du, Z., Wang, Z., Lai, H., Ding, M.,  
Yang, Z., Xu, Y., Zheng, W., Xia, X., et al. Glm-130b:  
An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414*, 2022.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M.,  
Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., Mi-  
haylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D.,  
Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer,  
L. Opt: Open pre-trained transformer language models,  
2022. URL <https://arxiv.org/abs/2205.01068>.

Zhao, R., Hu, Y., Dotzel, J., De Sa, C., and Zhang, Z. Improving neural network quantization without retraining using outlier channel splitting. In *International conference on machine learning*, pp. 7543–7552. PMLR, 2019.

## A Discussion on Weight-Only Quantization

In this work, we study W8A8 quantization so that we can utilize INT8 GEMM kernels to increase the throughput and accelerate inference. There is another line of work that only quantizes the weight of LLMs (e.g., GPTQ (Frantar et al., 2022)). It converts the quantized weights to FP16 on the fly for matmul during inference and can also lead to speed up due to the reduced data loading, especially for the generation stage with batch size 1.

We mainly compare our method with existing work on weight-activation quantization (i.e., W8A8) like (Dettmers et al., 2022; Yao et al., 2022; Wei et al., 2022) since they are under the same setting. Here we would like to give a short discussion about the weight-only quantization methods in LLM settings:

1. Firstly, we were trying to compare our method with GPTQ (Frantar et al., 2022) but found it difficult due to different implementations. GPTQ’s low-bit kernel<sup>¶</sup> only supports the generation stage with batch size 1 (i.e., only processing a single token at a time), and cannot support the context stage (widely used in different downstream tasks and chatbot) or batch-based setting. Furthermore, its low-bit kernel optimization only targets the OPT-175B model (as stated in the README). At the same time, our work utilizes FasterTransformer for serving large models, which may lead to an unfair advantage if we make a direct comparison.
2. GPTQ may perform better at handling a small number of input tokens (1 in its experiments) since the process is highly memory-bounded. In contrast, SmoothQuant may serve better with a batching setting or for the context stage (i.e., when the number of processed tokens is more significant). Nonetheless, some work shows that in production, we can improve the throughput of serving GPT models by  $37\times$  at similar latency with advanced batching (Yu et al., 2022). We believe in production, batching will be the future standard, and SmoothQuant will bring further improvement, even for the generation stage.
3. Applications like chatbots need to handle a long context length and potentially run under a batch setting. Due to the two factors, the memory size of the KV cache can no longer be ignored (as shown in (Pope et al., 2022)), the KV cache totals 3TB given batch size 512 and context length 2048, which is  $3\times$  larger than the model weights). In this case, quantization of activation can also help reduce the memory cost from storing the KV cache.

<sup>¶</sup><https://github.com/IST-DASLab/gptq>

4. Finally, we think the two settings are somewhat orthogonal. We believe we can integrate GPTQ’s method for a better weight quantization and potentially achieve W4A4 quantization, which will lead to even better hardware efficiency (INT4 instructions are supported on NVIDIA’s Hopper GPU architecture). We leave this exploration to future work.

# SpecInfer: Accelerating Generative Large Language Model Serving with Speculative Inference and Token Tree Verification

Xupeng Miao<sup>♣</sup> Gabriele Oliaro<sup>♣</sup> Zhihao Zhang<sup>♣</sup> Xinhao Cheng<sup>♣</sup> Zeyu Wang  
 Rae Ying Yee Wong Alan Zhu Lijie Yang Xiaoxiang Shi<sup>◊</sup> Chunan Shi<sup>†</sup>  
 Zhuoming Chen Daiyaan Arfeen Reyna Abhyankar<sup>‡</sup> Zhihao Jia

CMU <sup>◊</sup>Shanghai Jiao Tong University <sup>†</sup>Peking University <sup>‡</sup>UCSD

## Abstract

The high computational and memory requirements of generative large language models (LLMs) make it challenging to serve them quickly and cheaply. This paper introduces SpecInfer, an LLM serving system that accelerates generative LLM inference with speculative inference and token tree verification. A key insight behind SpecInfer is to combine various collectively boost-tuned small language models to jointly predict the LLM’s outputs; the predictions are organized as a token tree, whose nodes each represent a candidate token sequence. The correctness of all candidate token sequences represented by a token tree is verified against the LLM in parallel using a novel tree-based parallel decoding mechanism. SpecInfer uses an LLM as a token tree verifier instead of an incremental decoder, which significantly reduces the end-to-end latency and computational requirement for serving generative LLMs while provably preserving model quality. Our evaluation shows that SpecInfer outperforms existing LLM serving systems by 1.3-2.4× for distributed LLM inference and by 2.6-3.5× for offloading-based LLM inference, while preserving the same generative performance. SpecInfer is publicly available at <https://github.com/flexflow/FlexFlow/tree/inference>

## 1. Introduction

Generative large language models (LLMs), such as ChatGPT [3] and GPT-4 [30], have demonstrated remarkable capabilities of creating natural language texts across various domains, including summarization, instruction following, and question answering [54, 26]. However, it is challenging to quickly and cheaply serve these LLMs due to their large volume of parameters, complex architectures, and high computational requirements. For example, the GPT-3 architecture has 175 billion parameters, which require more than 16 NVIDIA 40GB A100 GPUs to store in single-precision floating points, and take several seconds to serve a single inference request [3].

An LLM generally takes as input a sequence of tokens, called *prompt*, and generates subsequent tokens one at a time, as shown in Figure 1a. The generation of each token in the sequence is conditioned on the input prompt and previously

generated tokens and does not consider future tokens. This approach is also called *autoregressive* decoding because each generated token is also used as input for generating future tokens. This dependency between tokens is crucial for many NLP tasks that require preserving the order and context of the generated tokens, such as text completion [53].

Existing LLM systems generally use an *incremental decoding* approach to serving a request where the system computes the activations for all prompt tokens in a single step and then iteratively decodes *one* new token using the input prompt and all previously generated tokens. This approach respects data dependencies between tokens, but achieves suboptimal runtime performance and limited GPU utilization, since the degree of parallelism within each request is greatly limited in the incremental phase. In addition, the attention mechanism of Transformer [46] requires accessing the keys and values of all previous tokens to compute the attention output of a new token. To avoid recomputing the keys and values for all preceding tokens, today’s LLM systems use a caching mechanism to store their keys and values for reuse in future iterations. For long-sequence generative tasks (e.g., GPT-4 supports up to 32K tokens in a request), caching keys and values introduce significant memory overhead, which prevents existing systems from serving a large number of requests in parallel due to the memory requirement of caching their keys and values.

This paper introduces SpecInfer, an LLM serving system that improves the end-to-end latency and computational efficiency of generative LLM inference with *speculative inference* and *token tree verification*. A key insight behind SpecInfer is to combine various collectively boost-tuned small speculative models (SSMs) to jointly predict the LLM’s output and use the LLM to verify *all* predictions, which allows SpecInfer to opportunistically decode multiple tokens in an LLM decoding step, while existing incremental decoding approach only generates one token in each step. The idea of *speculative execution* has been widely deployed in a variety of optimization tasks in computer architecture and systems, including branch prediction in modern pipelined processors and value prediction for pre-fetching memory and files [14, 40]. Leveraging speculative inference for accelerating LLMs, however, requires addressing two unique challenges. Next, we elaborate on these challenges and the main ideas SpecInfer uses to

<sup>♣</sup> Contributed equally.

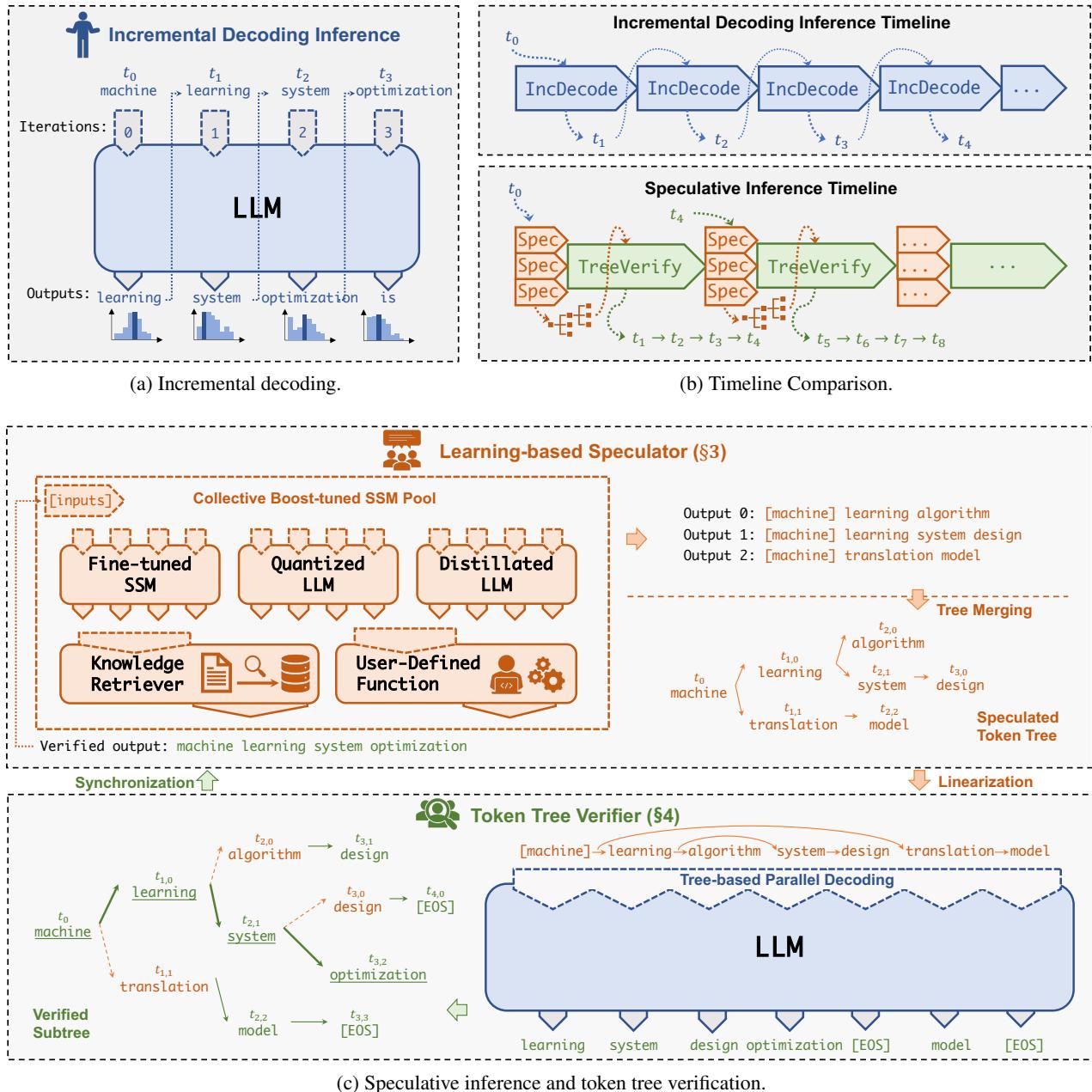


Figure 1: Comparing the incremental decoding approach used by existing LLM serving systems and the speculative inference and token tree verification approach used by SpecInfer.

address them.

First, speculative inference requires exploring a significantly larger search space than prior speculative techniques due to two reasons: (1) modern LLMs generally involve very large vocabularies, and (2) maximizing speculative performance requires predicting multiple future tokens (instead of just the next token). For example, all LLMs in the OPT model family consider 50,272 different possible tokens in their vocabulary, while SpecInfer can correctly predict the next 4 tokens on average. Achieving this goal requires considering a search

space of  $50272^4 \approx 6 \times 10^{18}$  different combinations of tokens.

To maximize the speculative performance, SpecInfer leverages existing distilled, quantized, and/or pruned variants of an LLM, which we call *small speculative models* (SSMs), to guide speculation. A key challenging of using SSMs for speculative inference is that the alignment between an SSM and an LLM is inherently bounded by the model capacity gap, since an SSM is generally 100-1000 $\times$  smaller than an LLM. Instead of using a single SSM for speculation, SpecInfer introduces *collective boost-tuning* to cooperatively fine-tune a

set of SSMs by aligning their *aggregated* prediction with the LLM, and combines these boost-tuned SSMs to jointly predict the LLM’s output.

A second challenge SpecInfer must address is verifying the speculated tokens. Many LLM applications perform *stochastic sampling*, which samples the next token from a probability distribution instead of deterministically generating a token. To preserve an LLM’s generative performance, SpecInfer must guarantee that its speculative inference and token tree verification procedure generates the next token by following the *exact same* probability distribution as incremental decoding. To achieve this goal, we propose *multi-step speculative sampling*, a new sampling approach for SSMs that guarantees equivalence while maximizing the number of speculated tokens that can be verified. To minimize the cost of token tree verification, SpecInfer introduces a *tree-based parallel decoding* mechanism that can *simultaneously* verify all tokens of a token tree against the LLM’s output in a *single* LLM decoding step.

By leveraging speculative inference and token tree verification, SpecInfer can accelerate both distributed LLM inference across multiple GPUs and offloading-based LLM inference on one GPU. Our evaluation shows that SpecInfer outperforms existing LLM serving systems by 1.3-2.4 $\times$  for distributed LLM inference and by 2.6-3.5 $\times$  for offloading-based LLM inference, while preserving the same generative performance.

## 2. SpecInfer’s Overview

---

**Algorithm 1** The incremental decoding algorithm used in existing LLM serving systems.

---

```

1: Input: A sequence of input tokens  $\mathcal{I}$ 
2: Output: A sequence of generated tokens
3:  $\mathcal{S} = \mathcal{I}$ 
4: while true do
5:    $t = \text{DECODE}(\text{LLM}, \mathcal{S})$ 
6:    $\mathcal{S}.\text{append}(t)$ 
7:   if  $t = \langle \text{EOS} \rangle$  then
8:     Return  $\mathcal{S}$ 
```

---

Figure 1 shows an overview of SpecInfer in comparison with incremental decoding. SpecInfer includes a *learning-based speculator* that takes as input a sequence of tokens, and produces a *speculated token tree*. The goal of the speculator is to predict the LLM’s output by maximizing the overlap between the speculated token tree and the tokens generated by the LLM using incremental decoding. As shown at the top of Figure 1c, the speculator combines multiple distilled and/or pruned versions of the LLM, which we call small speculative models (SSMs).

There are several ways to prepare SSMs for speculative inference. First, modern LLMs generally have many much smaller architectures pre-trained together with the LLM using the same datasets. For example, in addition to the OPT-175B model with 175 billion parameters, the OPT model family also

---

**Algorithm 2** The speculative inference and token tree verification algorithm used by SpecInfer. SPECULATE takes the current token sequence  $\mathcal{S}$  as an input and generates a speculated token tree  $\mathcal{N}$ . TREEPARALLELDECODE generates a token  $\mathcal{O}(u)$  for each node  $u \in \mathcal{N}$ . VERIFYGREEDY and VERIFYSTOCHASTIC examine the speculated token tree  $\mathcal{N}$  against the LLM’s output  $\mathcal{O}$  and produces a sequence of verified tokens  $\mathcal{V}$  using greedy or stochastic sampling, respectively.

---

```

1: Input: A sequence of input tokens  $\mathcal{I}$ 
2: Output: A sequence of generated tokens
3:  $\mathcal{S} = \mathcal{I}$ 
4: while true do
5:    $\mathcal{N} = \text{SPECULATE}(\mathcal{S})$ 
6:    $\mathcal{O} = \text{TREEPARALLELDECODE}(\text{LLM}, \mathcal{N})$ 
7:   if use greedy decoding then
8:      $\mathcal{V} = \text{VERIFYGREEDY}(\mathcal{O}, \mathcal{N})$ 
9:   else
10:     $\mathcal{V} = \text{VERIFYSTOCHASTIC}(\mathcal{O}, \mathcal{N})$ 
11:   for  $t \in \mathcal{V}$  do
12:      $\mathcal{S}.\text{append}(t)$ 
13:     if  $t = \langle \text{EOS} \rangle$  then
14:       return  $\mathcal{S}$ 
15:
16: function VERIFYGREEDY( $\mathcal{O}, \mathcal{N}$ )
17:    $\mathcal{V} = \emptyset, u \leftarrow$  the root of token tree  $\mathcal{N}$ 
18:   while  $\exists v \in \mathcal{N}. p_v = u$  and  $t_v = \mathcal{O}(u)$  do
19:      $u = v$ 
20:      $\mathcal{V}.\text{append}(t_v)$ 
21:    $\mathcal{V}.\text{append}(\mathcal{O}(u))$ 
22:   return  $\mathcal{V}$ 
23:
24: function VERIFYSTOCHASTIC( $\mathcal{O}, \mathcal{N}$ )
25:    $\mathcal{V} = \emptyset, u \leftarrow$  the root of token tree  $\mathcal{N}$ 
26:   while  $u$  is a non-leaf node do
27:      $\mathcal{H} = \text{child}(u)$             $\triangleright$  The set of child nodes for  $u$ 
28:     while  $\mathcal{H}$  is not empty do
29:        $s \sim \text{rand}(\mathcal{H}), r \sim U(0, 1), x_s = \mathcal{H}[s]$ 
30:       if  $r \leq P(x_s | u, \Theta_{\text{LLM}}) / P(x_s | u, \Theta_{\text{SSM}_s})$  then
31:          $\triangleright$  Token  $x_s$  passes verification.
32:          $\mathcal{V}.\text{append}(x_s)$ 
33:          $u = s$ 
34:         break
35:       else
36:          $\triangleright$  Normalize the residual  $P(x | u, \Theta_{\text{LLM}})$ 
37:          $P(x | u, \Theta_{\text{LLM}}) := \text{norm}(\max(0, P(x | u, \Theta_{\text{LLM}}) - P(x | u, \Theta_{\text{SSM}_s})))$ 
38:          $\mathcal{H}.\text{pop}(s)$ 
39:     if  $\mathcal{H}$  is empty then
40:       break
41:      $\triangleright$  All SSMs fail verification; sample the next token
42:      $x_{\text{next}} \sim P(x | u, \Theta_{\text{LLM}})$ 
43:      $\mathcal{V}.\text{append}(x_{\text{next}})$ 
44:   return  $\mathcal{V}$ 
```

---

includes OPT-125M and OPT-350M, two variants with 125 million and 350 million parameters, which were pre-trained using the same datasets as OPT-175B [55]. These pre-trained small models can be directly used as SSMs. Second, to maximize the coverage of speculated token trees, SpecInfer also introduces a fine-tuning technique called *collective boost-tuning* to cooperatively fine-tune a set of SSMs by aligning their aggregated prediction with the LLM’s using adaptive boosting [13].

SpecInfer’s usage of the LLM is also different from that of existing LLM serving systems. Instead of using the LLM as an incremental decoder that predicts the next single token, SpecInfer uses the LLM as a token tree verifier that verifies a speculated token tree against the LLM’s output. For each token, SpecInfer computes its activations by considering all of its ancestors in the token tree as its preceding tokens. For example, in Figure 1c, the attention output of the token  $t_{3,0}$  is calculated based on sequence  $(t_0, t_{1,0}, t_{2,1}, t_{3,0})$ , where  $t_0$ ,  $t_{1,0}$ , and  $t_{2,1}$  are  $t_{3,0}$ ’s ancestors in the token tree. SpecInfer includes a novel tree-based parallel decoding algorithm to simultaneously verify *all* tokens of a token tree in a single LLM decoding step.

SpecInfer’s speculative inference and token tree verification provide two key advantages over the incremental decoding approach of existing LLM inference systems.

**Reduced memory accesses to LLM parameters.** The performance of generative LLM inference is largely limited by GPU memory accesses. In the existing incremental decoding approach, generating a single token requires accessing all parameters of an LLM. The problem is exacerbated for offloading-based LLM inference systems, which use limited computational resources such as a single commodity GPU to serve LLMs by utilizing CPU DRAM and persistent storage to save model parameters and loading these parameters to GPU’s high bandwidth memory (HBM) for computation. Compared to the incremental decoding approach, SpecInfer significantly reduces accesses to LLM parameters whenever the overlap between a speculated token tree and the LLM’s actual output is not empty. Reduced accesses to GPU device memory and reduced data transfers between GPU and CPU memory can also directly translate to decreased energy consumption, since accessing GPU HBM consumes two or three orders of magnitude more energy than floating point arithmetic operations.

**Reduced end-to-end inference latency.** Serving LLMs suffers from long end-to-end inference latency. For example, the GPT-3 architecture includes 175 billion parameters and requires many seconds to serve a request. In the existing incremental decoding approach, the computation for generating each token depends on the keys and values of all previously generated tokens, which introduces sequential dependencies between tokens and requires modern LLM serving systems to serialize the generation of different tokens for each request. In SpecInfer, LLMs are used as a verifier that takes a speculated

token tree as an input and can simultaneously examine *all* tokens in the token tree by making a single verification pass over the LLM. This approach enables parallelization across different tokens in a single request and reduces the LLM’s end-to-end inference latency.

### 3. Learning-based Speculator

One major contribution of SpecInfer is the design and implementation of the speculator. As more accurate speculation can lead to speculated token trees with longer matching lengths, which in turn results in fewer LLM verification steps, improving the quality of the speculator is crucial. To this end, SpecInfer introduces collective boost-tuning, a novel fine-tuning technique that aligns the aggregated prediction of a set of SSMs with the LLM’s output using adaptive boosting.

As identified in prior work [25, 42], a key limitation of using a single SSM for speculative inference is that the alignment between an SSM and LLM is inherently bounded by the model capacity gap between the two models. Our preliminary exploration shows that using a larger model achieves better speculative performance but introduces additional memory overhead and inference latency.

Consequently, SpecInfer uses an unsupervised approach to collectively fine-tune a pool of SSMs to align their outputs with that of the LLM by leveraging adaptive boosting [13], as shown in Figure 2. SpecInfer’s SSMs are used to predict the next few tokens that an LLM will generate, therefore SpecInfer uses general text datasets (e.g., the OpenWebText corpus [15] in our evaluation) to adaptively align the aggregated output of multiple SSMs with the LLM in a fully unsupervised fashion. In particular, we convert a text corpus into a collection of prompt samples and use the LLM to generate a token sequence for each prompt. SpecInfer first fine-tunes one SSM at a time to the fullest and marks all prompt samples where the SSM and LLM generate identical subsequent tokens. Next, SpecInfer filters all marked prompt samples and uses all remaining samples in the corpus to fine-tune the next SSM to the fullest. By repeating this process for every SSM in the pool, SpecInfer obtains a diverse set of SSMs whose aggregated output largely overlaps with the LLM’s output on the training corpus. All SSMs have roughly identical inference latency, and therefore running all SSMs on different GPUs in parallel does not increase the latency of speculative inference compared to using a single SSM. Note that using multiple SSMs increases the memory overhead for storing their parameters on GPUs. However, our evaluation shows that SpecInfer can achieve significant performance improvement by using SSMs 100-1000 $\times$  smaller than the LLM, making the overhead of hosting these SSMs negligible. In our evaluation, we perform collective boost-tuning offline on publicly available datasets.

### 4. Token Tree Verifier

This section introduces SpecInfer’s *token tree verifier*, which takes as input a token tree generated by the speculator and

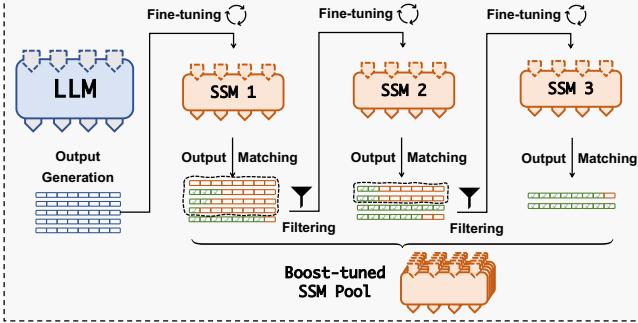


Figure 2: Illustrating SpecInfer’s collective boost-tuning technique. When using a single SSM to generate token trees, SpecInfer can verify 2.6 tokens on average in each LLM decoding step. This is due to the misalignment between an SSM and LLM on the first four token sequences. By collectively boost-tuning three SSMs, the average number of verified tokens per LLM decoding step is improved to 7.2.

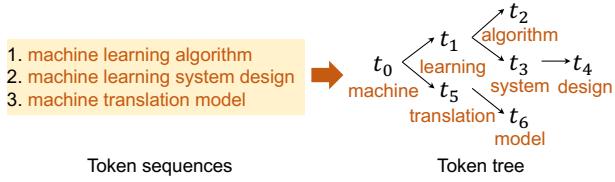


Figure 3: Merging three sequences of tokens (left) into a token tree (right), whose node each identifies a token sequence. For example, node  $t_4$  represents a token sequence  $[t_0, t_1, t_3, t_4]$  that translates to “machine learning system design”.

verifies the correctness of its tokens against an LLM’s output.

**Definition 4.1** (Token Tree). A token tree  $\mathcal{N}$  is a tree structure, where each node  $u \in \mathcal{N}$  is labelled with a token  $t_u$ , and  $p_u$  represents  $u$ ’s parent node in the token tree. For each node  $u$ ,  $S_u$  represents a sequence of tokens identified by concatenating  $S_{p_u}$  and  $\{t_u\}$ <sup>1</sup>.

SpecInfer uses a *token tree* to store the results generated by the learning-based speculator. Specifically, SpecInfer receives multiple token sequences generated by different SSMs, each of which can be considered as a token tree (with linear tree structure), and merges them into a single token tree.

**Definition 4.2** (Token Tree Merge).  $\mathcal{M}$  is the tree merge of  $m$  token trees  $\{\mathcal{N}_i\}$  ( $1 \leq i \leq m$ ) if and only if  $\forall 1 \leq i \leq m, \forall u \in \mathcal{N}_i, \exists v \in \mathcal{M}$  such that  $S_v = S_u$  and vice versa.

Intuitively, each token tree represents a set of token sequences. Merging multiple token trees produces a new tree that includes all token sequences of the original trees. For example, Figure 3 shows the token tree derived by merging three sequences of tokens. Each token sequence is identified by a node in the merged token tree.

A key idea behind the design of SpecInfer is *simultaneously* verifying all sequences of a token tree against the original

<sup>1</sup>For the root node  $r$ ,  $S_r$  represents the token sequence  $\{t_r\}$ .

LLM’s output by making a *single* pass over the LLM’s parameters. This functionality allows SpecInfer to opportunistically decode *multiple* tokens (instead of a single token in incremental decoding), resulting in reduced memory accesses to the LLM’s parameters. A challenge SpecInfer must address in token tree verification is efficiently computing the attention scores for *all* sequences of a token tree. To this end, we introduce *tree attention*, which generalizes the Transformer-based attention mechanism [46] from sequence to tree structure. In addition, we develop a *tree-based parallel decoding* mechanism that can decode *all* tokens in a tree token in parallel.

Section 4.1 and Section 4.2 describe tree attention and tree-based parallel decoding. Section 4.3 introduces the mechanism SpecInfer uses to verify a token tree against the LLM’s output.

#### 4.1. Tree Attention

Transformer-based language models use the attention mechanism to reason about sequential information [46]. LLMs generally use decoder-only, multi-head self-attention, which takes a single input tensor  $X$  and computes an output tensor  $O$  via scaled multiplicative formulations as follows.

$$Q_i = X \times W_i^Q, \quad K_i = X \times W_i^K, \quad (1)$$

$$V_i = X \times W_i^V, \quad A_i = \frac{(Q_i \times K_i^T)}{\sqrt{d}}, \quad (2)$$

$$H_i = \text{softmax}(\text{mask}(A_i)) V_i, \quad O = (H_1, \dots, H_h) W^O \quad (3)$$

where  $Q_i$ ,  $K_i$ , and  $V_i$  denote the query, key, and value tensors of the  $i$ -th attention head ( $1 \leq i \leq h$ ),  $W_i^Q$ ,  $W_i^K$ , and  $W_i^V$  are the corresponding weight matrices.  $A_i$  is an  $l \times l$  matrix that represents the attention scores between different tokens in the input sequence, where  $l$  is the sequence length. To preserve causality when generating tokens (i.e., a token in the sequence should not affect the hidden states of any preceding tokens), the following causal mask function is applied:

$$\text{mask}(A)_{jk} = \begin{cases} A_{jk} & j \geq k \\ -\infty & j < k \end{cases}. \quad (4)$$

Intuitively, when computing the attention output of the  $j$ -th token in the sequence, all subsequent tokens should have an attention score of  $-\infty$  to indicate that the subsequent tokens will not affect the attention output of the  $j$ -th token<sup>2</sup>. In Equation 3,  $H_i$  represents the output of the  $i$ -th attention head, and  $W^O$  is a weight matrix used for computing the final output of the attention layer.

Note that the attention mechanism described above applies only to a sequence of tokens. To address this issue, we generalize the attention mechanism to arbitrary tree structures.

**Definition 4.3** (Tree Attention). For a token tree  $\mathcal{N}$  and an arbitrary node  $u \in \mathcal{N}$ , its tree attention is defined as the output of

<sup>2</sup>Note that we use  $-\infty$  (instead of 0) to guarantee that the softmax’s output is 0 for these positions.

computing the original Transformer-based sequence attention on  $S_u$  (i.e., the token sequence represented by  $u$ ):

$$\text{TREEATTENTION}(u) = \text{ATTENTION}(S_u) \forall u \in \mathcal{N} \quad (5)$$

For a given set of token sequences, since each sequence  $S$  is covered by a node of the merged token tree, performing tree attention on the token tree allows SpecInfer to obtain the attention output for *all* token sequences. Note that the semantic of SpecInfer’s tree attention is different from prior tree-structured attention work [28], which we discuss in Section 7.

## 4.2. Tree-based Parallel Decoding

This section describes SpecInfer’s *tree-based parallel decoding* mechanism for computing tree attention for *all* tokens in a token tree *in parallel*. A key challenge SpecInfer must address in computing tree attention is managing *key-value cache*. In particular, the attention mechanism of Transformer [46] requires accessing the keys and values of all preceding tokens to compute the attention output of each new token, as shown in Equation 3. To avoid recomputing these keys and values, today’s LLM inference systems generally cache the keys and values of all tokens for reuse in future iterations, since the causal relation guarantees that a token’s key and value remain unchanged in subsequent iterations (i.e.,  $\text{mask}(A)_{jk} = -\infty$  for any  $j < k$ ). However, when computing tree attention, different sequences in a token tree may include conflicting key-value caches. For example, for the speculated token tree in Figure 4, two token sequences  $(t_2, t_3, t_4, t_5)$  and  $(t_2, t_3, t_8, t_9)$  have different keys and values for the third and fourth positions.

A straightforward approach to supporting key-value cache is employing the sequence-based decoding of existing LLM inference systems and using a different key-value cache for each sequence of a token tree, as shown on the left of Figure 4. However, this approach is computationally very expensive and involves redundant computation, since two token sequences sharing a common prefix have the same attention outputs for the common prefix due to the causal mask in Equation 3. In addition, launching one kernel for each token sequence introduces additional kernel launch overhead.

SpecInfer introduces two key techniques to realize tree-based parallel decoding.

**Depth-first search to update key-value cache.** Instead of caching the keys and values for individual token sequences of a token tree, SpecInfer reuses the same key-value cache across all token sequences by leveraging a *depth-first search* mechanism to traverse the token tree, as shown in Figure 4, where SpecInfer visits  $t_2, t_3, \dots, t_9$  by following a depth-first order to traverse the token tree and update the shared key-value cache. This approach allows SpecInfer to maintain the correct keys and values for all preceding tokens when computing the attention output of a new token.

**Topology-aware causal mask.** A straightforward approach to computing tree attention is calculating the tree attention

output for individual tokens by following the depth-first order described earlier. However, this approach would result in high GPU kernel launch overhead since each kernel only computes tree attention for one token sequence. In addition, executing these kernels in parallel requires additional GPU memory to store their key-value caches separately due to cache conflict. A key challenge that prevents SpecInfer from batching multiple tokens is that the attention computation for different tokens requires different key-value caches and therefore cannot be processed in parallel.

We introduce *topology-aware causal mask* to fuse tree attention computation of all tokens in a single kernel. To batch attention computation, SpecInfer uses a tree topology instead of the original sequence topology to store the keys and values of all tokens in a token tree in the key-value cache. For example, to compute tree attention for the speculated token tree shown in Figure 4, SpecInfer takes both verified tokens (i.e.,  $t_2$ ) and all speculated tokens (i.e.,  $t_3, t_4, \dots, t_9$ ) as inputs. This approach allows SpecInfer to fuse the attention computation into a single kernel but also results in attention scores that violate the causal dependency (e.g.,  $t_7$ ’s attention computation uses all previous tokens, including  $t_5$  which is not in  $t_7$ ’s token sequence). To fix the attention scores for these pairs, SpecInfer updates the causal mask based on the token tree’s topology. This approach computes the exact same attention output as incremental decoding, while resulting in much fewer kernel launches compared to existing sequence-based decoding mechanism.

## 4.3. Token Verification

For a given speculated token tree  $\mathcal{N}$ , SpecInfer uses tree-based parallel decoding (see Section 4.2) to compute its tree attention and generate an output tensor  $\mathcal{O}$  that includes a token for each node  $u \in \mathcal{N}$ . Next, SpecInfer’s *token tree verifier* examines the correctness of speculated tokens against the LLM. SpecInfer supports both greedy and stochastic sampling as shown in Algorithm 2.

**Greedy sampling.** Many LLM applications generate tokens using *greedy sampling*, which greedily selects the token with the highest likelihood in each decoding step. The VERIFYGREEDY function in Algorithm 2 shows how SpecInfer verifies a speculated token tree  $\mathcal{N}$  with greedy sampling. SpecInfer starts from the root of  $\mathcal{N}$  and iteratively examines a node’s speculated results against the LLM’s original output. For a node  $u \in \mathcal{N}$ , SpecInfer successfully speculates its next token if  $u$  includes a child node  $v$  (i.e.,  $p_v = u$ ) whose token matches the LLM’s output (i.e.,  $t_v = \mathcal{O}(u)$ ). In this case, SpecInfer finishes its verification for node  $u$  and moves on to examine its child  $v$ . When the node  $u$  does not include a child that contains the LLM’s output, SpecInfer adds  $\mathcal{O}(u)$  as a verified node in  $\mathcal{N}$  and terminates the verification process. Finally, all verified nodes are appended to the current generated token sequence  $\mathcal{V}$ . Token tree verification allows SpecInfer to opportunistically decode multiple tokens (instead

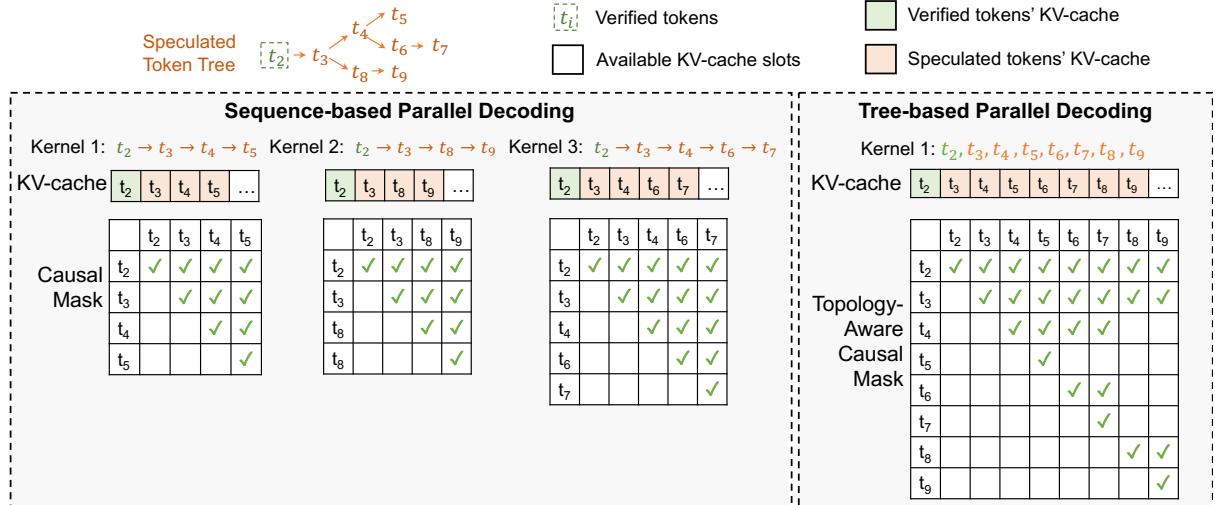


Figure 4: Comparing SpecInfer’s tree-based parallel decoding with existing sequence-based decoding.

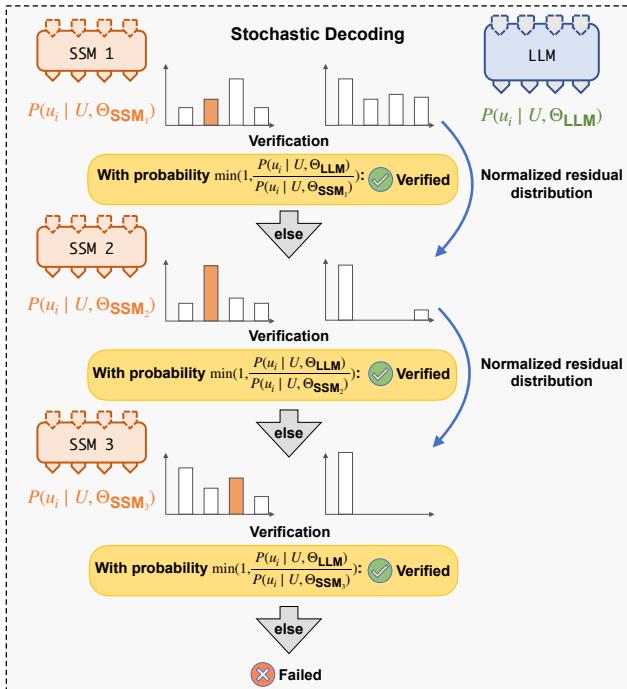


Figure 5: Illustrating SpecInfer’s multi-step speculative sampling mechanism for verifying LLMs with stochastic sampling.

of a single token in the incremental decoding approach), while preserving the same generative performance as incremental decoding.

**Stochastic sampling.** To improve the diversity of generated tokens, many LLM applications perform *stochastic sampling*, which samples a token from a probability distribution  $P(u_i | U, \Theta_{LLM})$ , where  $U = u_0, \dots, u_{i-1}$  are previously generated tokens,  $u_i$  is the next token to generate, and  $\Theta_{LLM}$  represents a parameterized LLM.

To verify a speculated token tree with stochastic sampling,

we introduce a *multi-step speculative sampling* algorithm to conduct verification, whose pseudocode code is shown in the VERIFYSTOCHASTIC function in Algorithm 2 and illustrated in Figure 5. Our method provably preserves an LLM’s generative performance as incremental decoding while optimizing the number of speculated tokens that can be verified. Theorem 4.4 proves its correctness.

**Theorem 4.4.** For a given LLM and  $m$  SSMs (i.e.,  $SSM_1, \dots, SSM_m$ ), let  $P(u_i | U; \Theta_{LLM})$  be the probability distribution of sampling a token using stochastic sampling, where  $U = u_0, \dots, u_{i-1}$  are previously generated tokens,  $u_i$  is the next token to generate,  $\Theta_{LLM}$  represents the parameterized LLM. Let  $P_{\text{SpecInfer}}(u_i | U; \Theta_{LLM}, \{\Theta_{SSM_j}\})$  be the probability distribution of sampling token  $u_i$  using SpecInfer’s multi-step speculative sampling (see the VERIFYSTOCHASTIC function in Algorithm 2), where  $\Theta_{SSM_j}$  is the  $j$ -th parameterized SSM. Then  $\forall U, u_i, \Theta_{LLM}, \Theta_{SSM_j}$  we have

$$P(u_i | U; \Theta_{LLM}) = P_{\text{SpecInfer}}(u_i | U; \Theta_{LLM}, \{\Theta_{SSM_j}\}) \quad (6)$$

A proof of this theorem is presented in Appendix A.

We acknowledge that a more straightforward approach to preserving the probability distribution of stochastic sampling is directly sampling the next token  $x \sim P(u_i | U; \Theta_{LLM})$  and examining whether  $x$  is a child node of  $u_{i-1}$  in the speculated token tree. We call this approach *naive sampling* and show that SpecInfer’s multi-step speculative sampling has a uniformly lower rejection probability than naive sampling.

**Theorem 4.5.** Let  $P(\text{reject} | MSS, U, \Theta_{LLM}, \{\Theta_{SSM_j}\})$  denote the probability of rejecting speculation following multi-step speculative sampling with abbreviation  $P(\text{reject} | MSS)$ , and  $P(\text{reject} | NS, U, \Theta_{LLM}, \{\Theta_{SSM_j}\})$  the probability of rejecting speculation following Naive Sampling (NS) with abbreviation  $P(\text{reject} | NS)$ . Then  $\forall U, \Theta_{LLM}, \{\Theta_{SSM_j}\}$ , we have

$$P(\text{reject} | MSS) \leq P(\text{reject} | NS)$$

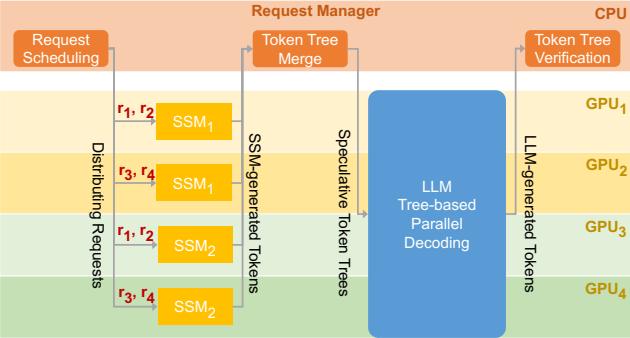


Figure 6: SpecInfer’s workflow for one iteration of speculative inference and token tree verification. SpecInfer uses data parallelism to serve SSMs, and combine tensor model parallelism and pipeline model parallelism for serving an LLM.

We present a proof of Theorem 4.5 in Appendix A.

## 5. System Design and Implementation

This section describes the design and implementation of SpecInfer’s distributed runtime system (§5.1 and §5.2), analyzes the computation and memory overheads of speculation and verification (§5.3), and introduces potential LLM applications that can benefit from SpecInfer’s techniques (§5.4).

### 5.1. SpecInfer’s Runtime Design

Figure 6 shows the workflow for one iteration of speculative inference and token tree verification. SpecInfer’s *request manager* receives LLM serving requests from end users and schedules these requests for serving by adapting the *iteration-level scheduling* policy from Orca [53]. Specifically, SpecInfer iteratively selects requests from a pool of pending requests and performs one iteration of speculative inference and token tree verification for the selected requests. Since SSMs are small and can fit in one GPU, SpecInfer equally distributes GPUs across SSMs and serves these SSMs using data parallelism. For example, Figure 6 shows how SpecInfer serves two SSMs and four requests (i.e.,  $r_1$ ,  $r_2$ ,  $r_3$ , and  $r_4$ ) on four GPUs. The SSM-generated tokens are sent back to the request manager, which produces a speculated token tree for each request using the tree merge algorithm introduced in §4.

SpecInfer serves an LLM using the hybrid parallelization strategy introduced in Megatron-LM [39], which uses tensor model parallelism for parallelizing each Transformer layer across GPUs within a node, and uses pipeline model parallelism for partitioning Transformer layers across nodes. All GPUs perform the tree-based parallel decoding (see §4.2) to compute tree attention scores and send the LLM-generated tokens back to the request manager, which finally verifies the speculated tokens against the LLM’s output (see §4.3).

Note that the overhead introduced by the request manager (i.e., request scheduling, token tree merge and verification) is negligible compared to the execution time of LLM inference. In addition, SpecInfer’s request manager and GPU workers

only communicate tokens and do not transfer the vector representations of these tokens, which again introduces negligible communication overheads.

### 5.2. SpecInfer’s Implementation

SpecInfer was implemented on top of FlexFlow [21, 45], a distributed multi-GPU runtime for DNN computation. FlexFlow exposes an API that allows users to define a DNN model in terms of its layers. Users can also provide a parallelization plan, specifying the degree of data, model, and pipeline parallelism of each layer. A DNN is represented as a computational graph where each node is a region of memory, and each edge is an operation on one or more regions. Operations can be represented using three levels of abstraction: layers, operators, and tasks. The FlexFlow compiler transforms the computational graph from the highest abstractions (layers) to the lowest (tasks). Tasks are also the unit of parallelization; they are non-preemptible, and are executed asynchronously.

### 5.3. Overhead of Speculation and Verification

SpecInfer accelerates generative LLM inference at the cost of memory and computation overheads. This section analyzes these overheads and show that they are generally one or two orders of magnitude smaller than the memory and computation cost of executing LLM inference.

**Memory overhead.** The memory overhead of SpecInfer’s speculation-verification approach comes from two aspects. First, in addition to serving an LLM, SpecInfer also needs to allocate memory for saving the parameters of one or multiple SSMs, which collectively speculate the LLM’s output. Our evaluation shows that SpecInfer can achieve significant performance improvement by using speculative models 100-1000 $\times$  smaller than the LLM. As a result, hosting each SSM increases the overall memory requirement by less than 1%. A second source of memory overhead comes from the token tree verification engine, which verifies an entire token tree instead of decoding a single token. Therefore, additional memory is needed for caching the keys and values, and storing the attention scores for all tokens. Due to the necessity for supporting very long sequence length in today’s LLM serving, we observe that the memory overhead associated with the token tree is negligible compared to the key-value cache. For example, GPT-4 supports processing up to 32K tokens in a single request; our evaluation shows that a token tree of size 40 already allows SpecInfer to match 3.7 tokens on average.

**Computation overhead.** Similarly, the computation overhead introduced by speculation and verification also comes from two aspects. First, SpecInfer needs to run multiple SSMs in the incremental-decoding mode to generate candidate token sequences. SpecInfer processes the SSMs in parallel across GPUs to minimize the latency for generating a speculated token tree. Second, SpecInfer verifies a token tree by computing the attention outputs for all token sequences of the

tree, most of which do not match the LLM’s output and therefore are unnecessary in the incremental-decoding inference. However, the key-value cache mechanism of existing LLM inference systems prevents them from serving a large number of requests in parallel, resulting in under-utilized computation resources on GPUs when serving LLMs in incremental decoding. SpecInfer’s token tree verification leverages these under-utilized resources and therefore introduces negligible runtime overhead compared to incremental decoding.

#### 5.4. Applications

Our speculative inference and token tree verification techniques can be directly applied to a variety of LLM applications. We identify two practical scenarios where LLM inference can significantly benefit from our techniques.

**Distributed generative LLM inference.** The memory requirements of modern LLMs exceed the capacity of a single compute node with one or multiple GPUs, and the current approach to addressing the high memory requirement is distributing the LLM’s parameters across multiple GPUs. For example, serving a single inference pipeline for GPT-3 with 175 billion parameters requires more than 16 NVIDIA A100-40GB GPUs to store the model parameters in single-precision floating points. Distributed generative LLM inference is largely limited by the latency to transfer intermediate activations between GPUs for each LLM decoding step. While SpecInfer’s approach does not directly reduce the amount of inter-GPU communications for LLM inference, SpecInfer verification mechanism can increase the communication granularity and reduce the number of LLM decoding steps.

**Offloading-based generative LLM inference.** Another practical scenario that can benefit from SpecInfer’s techniques is offloading-based generative LLM inference, which leverages CPU DRAM to store an LLM’s parameters and loads a subset of these parameters to GPUs for computation in a pipeline fashion [38]. By opportunistically verifying multiple tokens, SpecInfer can effectively reduce the number of LLM decoding steps and the overall communication between CPU DRAM and GPU HBM.

## 6. Evaluation

### 6.1. Experimental Setup

**LLMs.** To compare the runtime performance of SpecInfer against existing LLM serving systems, we evaluate these systems using two publicly available LLM families: OPT [55] and LLaMA [44]. More specifically, we select OPT-13B, OPT-30B, LLaMA-30B, and LLaMA-65B as the LLMs and collectively boost-tune SSMs from OPT-125M and LLaMA-160M. The pre-trained model parameters for the LLMs were directly acquired from their HuggingFace repositories [19]. We didn’t find a publicly available pre-trained version of small LLaMA models, and therefore trained a LLaMA-160M from

scratch for one epoch using the Wikipedia dataset [10] and part of the C4 dataset [35], which took approximately 275 hours on a single NVIDIA A100 GPU. We also used the OpenWebText Corpus [15] to collectively boost-tune multiple SSMs for speculative inference. Note that we only need to pre-train and collectively boost-tune SSMs once for each LLM model family before serving. Section 6.4 evaluates how collective boost-tuning can further improve speculation performance.

**Datasets.** We evaluate SpecInfer on five conversational datasets: Chatbot Instruction Prompts (CIP) [31], ChatGPT Prompts (CP) [27], WebQA [1], Alpaca [43, 33], and PIQA [2]. We only use the prompts/questions from these datasets to form our input prompts to simulate the real-world conversation trace.

**Platform.** The experiments were conducted on two AWS g5.12xlarge instances, each of which is equipped with four NVIDIA A10 24GB GPUs, 48 CPU cores, and 192 GB DRAM. Nodes are connected by 100 Gbps Ethernet.

For all experiments in Section 6.2 and Section 6.3, SpecInfer uses two SSMs and a speculation depth of 8, which achieves good performance in practice. Section 6.4 evaluates how different numbers of SSMs affect SpecInfer’s performance.

### 6.2. Distributed LLM Inference

We compare the end-to-end distributed LLM inference performance among SpecInfer, vLLM [24], HuggingFace Text Generation Inference (TGI) [18], and FasterTransformer [29] on OPT-30B, LLaMA-30B, and LLaMA-65B. For OPT-30B and LLaMA-30B, all systems serve the two LLMs in half-precision floating points across four A10 GPUs using tensor model parallelism. LLaMA-65B do not fit on four GPUs on a single node, therefore both FasterTransformer and SpecInfer serve it on eight A10 GPUs on two nodes by combining tensor model parallelism within each node and pipeline model parallelism across nodes. vLLM and HuggingFace TGI do not support pipeline model parallelism and cannot serve an LLM on multiple nodes. To rule out potential effects of our runtime implementation, we also evaluate SpecInfer using incremental decoding, which is achieved by sending an empty token tree to the verifier, so the verifier verifies exactly one token in each decoding step.

We use prompts from the five datasets described in Section 6.1. For each prompt, we let all systems generate up to 128 new tokens and report the average per-token latency in Figure 7. Note that SpecInfer may generate more than 128 new tokens since the verifier can verify multiple tokens in each iteration. In this case, we truncate SpecInfer’s output to 128 tokens. SpecInfer with incremental decoding achieves on-par performance as existing systems. This is likely because all systems use the same strategies to parallelize LLM inference across GPUs and use the same kernel libraries (i.e., cuDNN, cuBLAS, and cuTLOSS) to execute inference computation on GPUs. With speculative inference and token tree

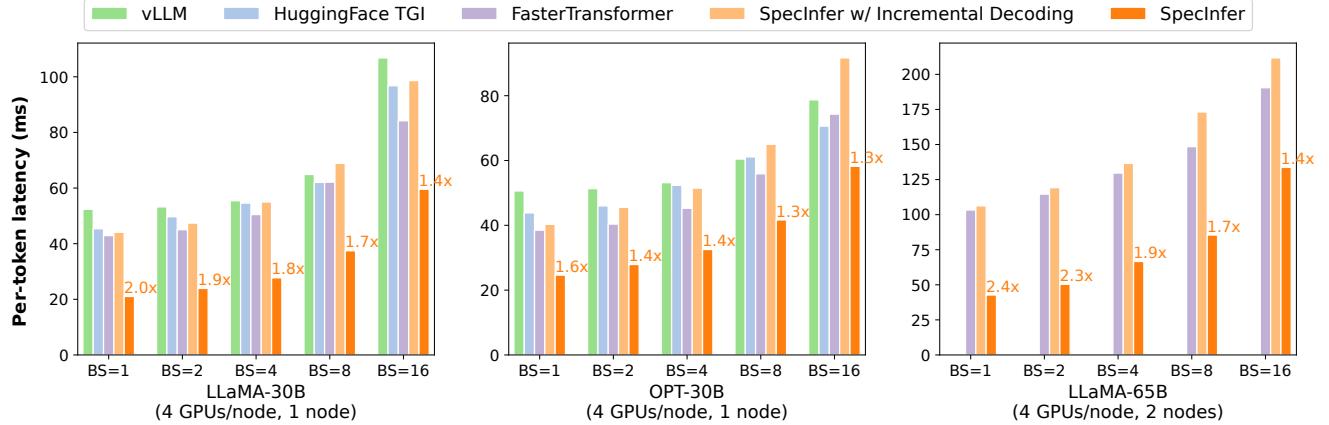


Figure 7: Comparing the end-to-end inference latency of incremental decoding and SpecInfer on five prompt datasets. We use LLaMA-7B as the LLM and all SSMS are derived from LLaMA-160M. The performance is normalized by incremental decoding, and the numbers on the SpecInfer bars indicate the speedups over incremental decoding.

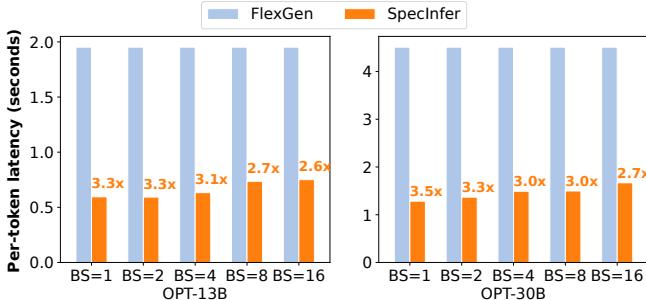


Figure 8: Comparing the end-to-end offloading-based inference latency of FlexGen and SpecInfer. Both FlexGen and SpecInfer perform model offloading to serve OPT-13B and OPT-30B models on a single 24GB A10 GPU.

verification, SpecInfer outperforms existing systems by 1.3–2.0× for single-node, multi-GPU inference and by 1.4–2.4× for multi-node, multi-GPU inference, while generating the exact same sequence of tokens as incremental decoding for all prompts. The performance improvement is realized by SpecInfer’s ability to verify multiple tokens in a single LLM decoding step.

Note that SpecInfer’s performance improvement over existing systems reduces as the batch size (i.e., number of concurrent requests) increases. This is because SpecInfer leverages spare GPU resources to perform tree-based parallel decoding while maintaining the same per-iteration latency as incremental decoding. A larger batch size introduces more parallelizable computation for incremental decoding, and thus less spare GPU resources that can be leveraged by SpecInfer. On the flip side, larger batch sizes also increase the end-to-end latency of each request, as shown in Figure 7. Overall, SpecInfer is most beneficial for *low-latency* LLM inference.

Table 1: Average number of tokens verified by SpecInfer in a decoding step. We used LLaMA-7B as the LLM and used different numbers of collectively boost-tuned SSMSs, all of which were derived from LLaMA-160M. The speculation length is 16 for all SSMSs.

# SSMSs	1	2	3	4	5
CIP	3.35	3.74	3.97	4.05	<b>4.11</b>
CP	2.71	3.14	3.32	3.45	<b>3.51</b>
WebQA	2.84	3.08	3.20	3.27	<b>3.31</b>
Alpaca	2.70	3.19	3.36	3.44	<b>3.49</b>
PIQA	2.98	3.21	3.36	3.44	<b>3.49</b>
Avg	2.92	3.27	3.44	3.53	<b>3.58</b>

### 6.3. Offloading-based LLM Inference

Another important application of SpecInfer is offloading-based LLM inference, where the system offloads an LLM’s parameters to CPU DRAM and loads a subset of these parameters to GPUs for inference computation in a pipeline fashion. We compare the end-to-end offloading-based LLM inference performance between SpecInfer and FlexGen [37] using a single 24GB A10 GPU and two LLMs (i.e., OPT-13B and OPT-30B), both of which exceed the memory capacity of an A100 GPU and requires offloading mechanism for serving. Both Collie and FlexGen retain all the parameter weights within the CPU memory. During computation, the demand weights are loaded from the CPU to the GPU. Figure 8 shows the results. Compared to FlexGen, SpecInfer reduces the per-token latency by 2.6–3.3×. Since offloading-based LLM inference is mostly bottlenecked by the communication between CPU DRAM and GPU HBM for loading an LLM’s parameters, SpecInfer’s improvement over existing systems is achieved by opportunistically verifying multiple tokens, which in turn reduces the number of LLM decoding steps and data transfers between CPUs and GPUs.

Table 2: Average number of tokens verified by SpecInfer in a decoding step. We used OPT-13B as the LLM and used different numbers of collectively boost-tuned SSMs, all of which were derived from OPT-125M. The speculation length is 16 for all SSMs.

# SSMs	1	2	3	4	5
CIP	3.00	3.39	3.52	3.58	<b>3.74</b>
CP	2.95	3.35	3.49	3.52	<b>3.68</b>
WebQA	2.51	2.92	3.04	3.09	<b>3.20</b>
Alpaca	3.33	3.89	4.06	4.17	<b>4.35</b>
PIQA	2.75	3.14	3.26	3.31	<b>3.43</b>
Avg	2.91	3.34	3.47	3.53	<b>3.68</b>

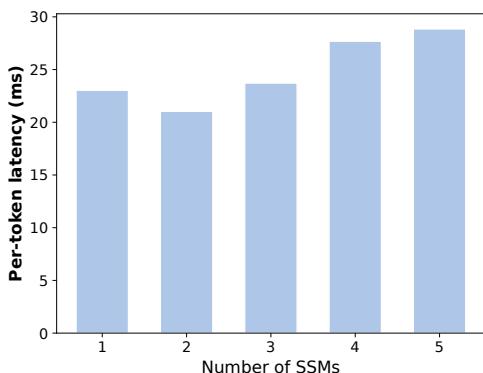


Figure 9: Comparing SpecInfer’s performance with different numbers of SSMs. We use LLaMA-30B as the LLM and the five collectively boost-tuned LLaMA-160M models as the SSMs.

#### 6.4. Collective Boost-Tuning

In this section, we evaluate the effectiveness of collective boost-tuning in terms of improving the average number of verified tokens in each LLM decoding step. For both the OPT and LLaMA experiments, we fine-tuned four SSMs over the OpenWebText Corpus using collective boost-tuning on top of the pre-trained OPT-125M and LLaMA-160M models, which provides a collection of five SSMs (including the base SSM) in each experiment. As shown in Table 1 and Table 2, the average number of tokens verified by SpecInfer in each LLM decoding step increases consistently across all five datasets due to better alignment between the LLM and our tuned collection of SSMs. More specifically, by using collectively boost-tuned SSMs, we have an overall improvement of 26.4% and 22.6% respectively, compared to using only a single pre-trained SSM.

We further study how different numbers of SSMs affect SpecInfer’s inference latency. In this experiment, we use LLaMA-60B as the LLM and the five collectively boost-tuned LLaMA-160M models as the SSMs. SpecInfer uses tensor model parallelism to serve LLaMA-30B across 4 GPUs and uses data parallelism to serve SSMs on different GPUs. Figure 9 shows the results. SpecInfer’s performance is insensitive to the number of SSMs for two reasons. First, SpecInfer uses

data parallelism to serve SSMs on different GPUs in parallel. As a result, adding more SSMs does not increase the speculation time and marginally increase the verification time (since SpecInfer needs to verify more tokens). Second, as shown in Table 2 and Table 1, the first several SSMs are critical to achieving high speculative performance and the average matching length increases slightly (i.e., from 3.27 to 3.58 and from 3.34 to 3.68 for OPT and LLaMA) as we introduce more SSMs.

## 7. Related Work

Transformer-based [46] LLMs have demonstrated significant potential in numerous human-level language modeling tasks by continuously increasing their sizes [34, 41, 9, 7]. As GPT-3 [3] becomes the first model to surpass 100B parameters, multiple LLMs ( $>100B$ ) have been released, including OPT-175B [55], Bloom-176B [36], and PaLM [7]. Recent work has proposed a variety of approaches to accelerating generative LLM inference, which can be categorized into two classes.

**Lossless acceleration.** Prior work has explored the idea of using an LLM as a verifier instead of a decoder to boost inference. For example, Yang et al. [51] introduced *inference with reference*, which leverages the overlap between an LLM’s output and the references obtained by retrieving documents, and checks each reference’s appropriateness by examining the decoding results of the LLM. Motivated by the idea of speculative execution in processor optimizations [4, 16], recent work proposed *speculative decoding*, which uses a small language model to produce a sequence of tokens and examines the correctness of these tokens using an LLM [25, 49, 42, 5, 22]. There are three key differences between SpecInfer and these prior works. First, instead of only considering a single sequence of tokens, SpecInfer generates and verifies a token tree, whose nodes each represent a unique token sequence. SpecInfer performs tree attention to compute the attention output of these token sequences in parallel and uses a novel tree-based decoding algorithm to reuse intermediate results shared across these sequences. Second, prior attempts generally consider a single small language model for speculation, which cannot align well with an LLM due to the model capacity gap between them. SpecInfer introduces collective boost-tuning to adapt different SSMs to align with an LLM under different scenarios, which largely increases the coverage of the speculated token trees produced by SpecInfer. Third, an additional challenge SpecInfer has to address is deciding the speculative configuration for a given request. SpecInfer leverages an important observation that the tokens generated by an LLM involve diverse difficulties to speculate, and uses a learning-based speculator to decide which SSMs to use and their speculative configurations.

Prior work has also introduced a variety of techniques to optimize ML computations on modern hardware platforms. For example, TVM [6] and Ansor [56] automatically generate

efficient kernels for a given tensor program. TASO [20] and PET [48] automatically discover graph-level transformations to optimize the computation graph of a neural architecture. SpecInfer’s techniques are orthogonal and can be combined with these systems to accelerate generative LLM computation, which we believe is a promising avenue for future work.

**Lossy acceleration.** BiLD [23] is a speculative decoding framework that uses a single SSM to accelerate LLM decoding. Unlike the systems mentioned above, the acceleration is lossy: speed comes at the cost of a possible degradation in the generated tokens. Another line of research leverages model compression to reduce LLM inference latency while compromising the predictive performance of the LLM. For example, prior work proposed to leverage weight/activation quantization of LLMs to reduce the memory and computation requirements of serving these LLMs [50, 12, 32, 52, 8]. Recent work further explores a variety of structured pruning techniques for accelerating Transformer-based architectures [11, 47, 17]. A key difference between SpecInfer and these prior works is that SpecInfer does not directly reduce the computation requirement for performing LLM inference, but instead reorganizing LLM inference computation in a more parallelizable way, which reduces memory accesses and inference latency at the cost of manageable memory and computation overheads.

**Tree-structured attention.** Nguyen et al. [28] introduced *tree-structured attention*, a technique that lets a Transformer model capture the hierarchical composition of input text by running the model on the text’s parse tree. It uses a one-on-one mapping to encode and decode the tree, so that the attention can process it. There are two key differences from SpecInfer’s tree-based decoding. First, SpecInfer uses a tree to combine candidate sequences to condense prefixes, whereas Nguyen et al. represent a single sequence with its parse tree. SpecInfer does not incorporate parse tree structure into the LLM, but accelerates inference by verifying decoded sequences in parallel. Second, SpecInfer’s attention outputs a token sequence, not a tree.

## 8. Conclusion

This paper introduces SpecInfer, an LLM serving system that accelerates generative LLM inference with speculative inference and token tree verification. A key insight behind SpecInfer is to combine various collectively boost-tuned versions of small language models to efficiently predict the LLM’s outputs, which are organized as a token tree and verified against the LLM in parallel using a tree-based parallel decoding mechanism. SpecInfer significantly reduces the memory accesses to the LLM’s parameters and the end-to-end LLM inference latency for both distributed and offloading-based LLM inference.

## Acknowledgement

We thank Tianqi Chen, Bohan Hou, and Hongyi Jin for thoughtful discussions and feedback on this work. This research is partially supported by NSF awards CNS-2147909, CNS-2211882, and CNS-2239351, two Amazon research awards, a Google faculty award, a Meta research award, and a Tang family endowment.

## References

- [1] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, Seattle, Washington, USA, October 2013. Association for Computational Linguistics.
- [2] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqq: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.
- [3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.
- [4] F Warren Burton. Speculative computation, parallelism, and functional programming. *IEEE Transactions on Computers*, 100(12):1190–1193, 1985.
- [5] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.
- [6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.
- [7] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.
- [8] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35:30318–30332, 2022.
- [9] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, pages 5547–5569. PMLR, 2022.
- [10] Wikimedia Foundation. Wikimedia downloads.
- [11] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot, 2023.
- [12] Elias Frantar, Saleh Ashkboos, Torsten Hoefer, and Dan Alistarh. Gptq: Accurate quantization for generative pre-trained transformers. In *International Conference on Learning Representations*, 2023.
- [13] Yoav Freund, Robert Schapire, and Naoki Abe. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771–780):1612, 1999.
- [14] Freddy Gabbay and Avi Mendelson. *Speculative execution based on value prediction*. Citeseer, 1996.
- [15] Aaron Gokaslan\*, Vanya Cohen\*, Ellie Pavlick, and Stefanie Tellex. Openwebtext corpus. <http://Skylion007.github.io/OpenWebTextCorpus>, 2019.
- [16] John L Hennessey and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [17] Itay Hubara, Brian Chmiel, Moshe Island, Ron Banner, Joseph Naor, and Daniel Soudry. Accelerated sparse neural training: A provable and efficient method to find n: m transposable masks. *Advances in Neural Information Processing Systems*, 34:21099–21111, 2021.

- [18] HuggingFace. Large language model text generation inference. <https://github.com/huggingface/text-generation-inference>. (Accessed on 08/09/2023).
- [19] Hugging Face Inc. Hugging face. <https://huggingface.co>, 2023.
- [20] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.
- [21] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning, SysML’19*, 2019.
- [22] Joao Gante. Assisted generation: a new direction toward low-latency text generation, 2023.
- [23] Sehoon Kim, Karttikeya Mangalam, Suhong Moon, John Canny, Jitendra Malik, Michael W. Mahoney, Amir Gholami, and Kurt Keutzer. Big little transformer decoder, 2023.
- [24] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. vllm: Easy, fast, and cheap llm serving with pagedattention. See <https://vllm.ai/> (accessed 9 August 2023), 2023.
- [25] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. *arXiv preprint arXiv:2211.17192*, 2022.
- [26] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021.
- [27] MohamedRashad. Chatgpt-prompts. <https://huggingface.co/datasets/MohamedRashad/ChatGPT-prompts>, 2023.
- [28] Xuan-Phi Nguyen, Shafiq Joty, Steven Hoi, and Richard Socher. Tree-structured attention with hierarchical accumulation. In *International Conference on Learning Representations*, 2020.
- [29] NVIDIA. Fastertransformer. <https://github.com/NVIDIA/FasterTransformer>. (Accessed on 08/09/2023).
- [30] OpenAI. Gpt-4 technical report, 2023.
- [31] Alessandro Palla. chatbot instruction prompts. [https://huggingface.co/datasets/alespalla/chatbot\\_instruction\\_prompts](https://huggingface.co/datasets/alespalla/chatbot_instruction_prompts), 2023.
- [32] Gunho Park, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. nuqmm: Quantized matmul for efficient inference of large-scale generative language models. *arXiv preprint arXiv:2206.09557*, 2022.
- [33] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. Instruction tuning with gpt-4. *arXiv preprint arXiv:2304.03277*, 2023.
- [34] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.
- [35] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv e-prints*, 2019.
- [36] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.
- [37] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.
- [38] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. High-throughput generative inference of large language models with a single gpu, 2023.
- [39] Mohammad Shoeybi, Mostafa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.
- [40] James E Smith. A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215, 1998.
- [41] Shaden Smith, Mostafa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhudesai, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.
- [42] Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. Blockwise parallel decoding for deep autoregressive models. *Advances in Neural Information Processing Systems*, 31, 2018.
- [43] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. [https://github.com/tatsu-lab/stanford\\_alpaca](https://github.com/tatsu-lab/stanford_alpaca), 2023.
- [44] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- [45] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efraim Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, Carlsbad, CA, July 2022. USENIX Association.
- [46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.
- [47] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 97–110. IEEE, 2021.
- [48] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54. USENIX Association, July 2021.
- [49] Heming Xia, Tao Ge, Si-Qing Chen, Furu Wei, and Zhifang Sui. Speculative decoding: Lossless speedup of autoregressive translation.
- [50] Guangxuan Xiao, Ji Lin, Mickael Seznec, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*, 2022.
- [51] Nan Yang, Tao Ge, Liang Wang, Binxing Jiao, Dexin Jiang, Linjun Yang, Rangan Majumder, and Furu Wei. Inference with reference: Lossless acceleration of large language models. *arXiv preprint arXiv:2304.04487*, 2023.
- [52] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in Neural Information Processing Systems*, 35:27168–27183, 2022.
- [53] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.
- [54] Haoyu Zhang, Jianjun Xu, and Ji Wang. Pretraining-based natural language generation for text summarization. *arXiv preprint arXiv:1902.09243*, 2019.
- [55] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuhui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.
- [56] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 863–879, 2020.

## A. Proof of the Theorems

**Theorem A.1.** For a given LLM and  $m$  SSMs (i.e.,  $SSM_1, \dots, SSM_m$ ), let  $P(u_i | U; \Theta_{LLM})$  be the probability distribution of sampling a token using stochastic sampling, where  $U = u_0, \dots, u_{i-1}$  are previously generated tokens,  $u_i$  is the next token to generate,  $\Theta_{LLM}$  represents the parameterized LLM. Let  $P_{\text{rmSpecInfer}}(u_i | U; \Theta_{LLM}, \{\Theta_{SSM_j}\})$  be the probability distribution of sampling token  $u_i$  using SpecInfer's multi-step speculative sampling (see the VERIFYSTOCHASTIC function in Algorithm 2), where  $\Theta_{SSM_j}$  is the  $j$ -th parameterized SSM. Then  $\forall U, u_i, \Theta_{LLM}, \Theta_{SSM_j}$  we have

$$P(u_i | U; \Theta_{LLM}) = P_{\text{SpecInfer}}(u_i | U; \Theta_{LLM}, \{\Theta_{SSM_j}\}) \quad (7)$$

*Proof.* It suffices to prove the equivalency for a single decoding step. Let  $u_{i-1}$  be the last token we have verified in the speculated token tree  $\mathcal{N}$ ,  $m$  be the number of  $u_{i-1}$ 's child nodes. In multi-step speculative sampling, SpecInfer iteratively verifies each child node and its SSM against the LLM. We define  $r_j = \sum_i \max(0, P(u_i | U, \Theta_{SSM_j}) - P(u_i | U, \Theta_{LLM}))$ ,  $\forall j \in [m]$  as the rejection probability at round  $j$ ,  $T_0 = P(u | U, \Theta_{LLM})$  and  $T_j = \frac{T_{j-1} - P(u | U, \Theta_{SSM_j})}{r_j}$ ,  $\forall j \in [m]$ . We further define

$$A_m = \max(0, T_m),$$

$$A_{j-1} = \min(P(u | U, \Theta_{SSM_j}), \max(T_{j-1}, 0)) + r_j A_j, \forall j \in [m]$$

Then following the VERIFYSTOCHASTIC sampling procedure stated in Algorithm 2, the probability of sampling  $u_i$  is:

$$P_{\text{SpecInfer}}(u_i | U; \Theta_{LLM}, \{\Theta_{SSM_j}\}) = A_0$$

Next, we will show that  $A_j = \max(0, T_j)$ ,  $\forall j \in [m]$  by backward induction. For the base case  $j = m$ , we have  $A_m = \max(0, T_m)$  by definition. Suppose  $A_j = \max(0, T_j)$  for  $j = n$ , we then have:

$$\begin{aligned} A_{n-1} &= \min(P(u | U, \Theta_{SSM_n}), \max(T_{n-1}, 0)) + r_n A_n \\ &= \min(P(u | U, \Theta_{SSM_n}), \max(T_{n-1}, 0)) + r_n \max(0, T_n) \\ &= \min(P(u | U, \Theta_{SSM_n}), \max(T_{n-1}, 0)) \\ &\quad + \max(0, T_{n-1} - P(u | U, \Theta_{SSM_n})) \\ &= \max(0, T_{n-1}) \end{aligned}$$

Thus, we have  $A_j = \max(0, T_j)$ ,  $\forall j \in [m]$ . Combining previous results, we have  $P_{\text{SpecInfer}}(u_i | U; \Theta_{LLM}, \{\Theta_{SSM_j}\}) = A_0 = \max(0, T_0) = P(u_i | U; \Theta_{LLM})$ , which concludes our proof. Notice that the overall rejection probability is  $\prod_{j=1}^m r_j$ .  $\square$

**Theorem A.2.** Let  $P(\text{reject} | MSS, U, \Theta_{LLM}, \{\Theta_{SSM_j}\})$  denote the probability of rejecting speculation following multi-step speculative sampling with abbreviation  $P(\text{reject} | MSS)$ ,

and  $P(\text{reject} | NS, U, \Theta_{LLM}, \{\Theta_{SSM_j}\})$  the probability of rejecting speculation following Naive Sampling (NS) with abbreviation  $P(\text{reject} | NS)$ . Then  $\forall U, \Theta_{LLM}, \{\Theta_{SSM_j}\}$ , we have

$$P(\text{reject} | MSS) \leq P(\text{reject} | NS)$$

We present the proof of Theorem 4.5 is presented below:

*Proof.* As before, we only need to prove the inequality for a single step as this suffices for proving over multiple steps. By the law of total probability, it suffices to show

$$P(\text{reject} \cap \mathcal{O}(u) = t | NS) \geq P(\text{reject} \cap \mathcal{O}(u) = t | MSS)$$

for all tokens  $t$ . Here  $\mathcal{O}(u)$  denotes the token selected by the according algorithm. Without loss of generality, we can fix token  $t$ . Let  $m$  be the number of child nodes.

NS rejects the speculation while selecting token  $t$  with probability  $P(t | U, \Theta_{LLM}) \prod_{j=1}^m (1 - P(t | U, \Theta_{SSM_j}))$  as this occurs when the LLM selects token  $t$ , but none of the  $m$  children do.

As for MSS, denote  $r_j$  as the rejection probability at round  $j$ . With intermediate results from Theorem 4.4, we may equivalently define  $A_0 = P(t | U, \Theta_{LLM})$  and  $A_j = \frac{A_{j-1} - \min(P(t | U, \Theta_{SSM_j}), A_{j-1})}{r_j} = \max(0, \frac{A_{j-1} - P(t | U, \Theta_{SSM_j})}{r_j})$ ,  $\forall j \in [m]$ . Note that for  $j \in [m]$ ,  $A_j$  is the probability of sampling  $t$  for the normalized residual distribution at the end of round  $j$ , and  $A_m$  is the probability of sampling  $t$  if the procedure rejects the speculation.

Thus, MSS rejects the speculation while selecting token  $t$  with probability  $A_m \prod_{j=1}^m r_j$ . We claim that this is upper bounded by  $\max(0, (P(t | U, \Theta_{LLM}) - P(t | U, \Theta_{SSM_1})) \prod_{j=2}^m (1 - P(t | U, \Theta_{SSM_j})))$ . We now case on whether there exists round  $k$  such that  $P(t | U, \Theta_{SSM_k}) \geq A_{k-1}$ .

**Case 1:**  $P(t | U, \Theta_{SSM_k}) \geq A_{k-1}$  for some  $k$ . Then  $A_\ell = 0$  for all  $\ell \geq k$ , and  $A_m \prod_{j=1}^m r_j = 0$ , and our upper bound holds.

**Case 2:**  $P(t | U, \Theta_{SSM_k}) < A_{k-1}$  for all rounds  $k$ . Then  $A_j = \frac{A_{j-1} - P(t | U, \Theta_{SSM_j})}{r_j}$ ,  $\forall j \in [m]$ . A lower bound for  $r_j$  is  $A_{j-1} - P(t | U, \Theta_{SSM_j})$ . We show

$$A_m \prod_{j=1}^m r_j \leq (P(t | U, \Theta_{LLM}) - P(t | U, \Theta_{SSM_1})) \prod_{j=2}^m (1 - P(t | U, \Theta_{SSM_j}))$$

To do this, we prove the stronger claim that

$$A_n \prod_{j=1}^n r_j \leq (P(t | U, \Theta_{LLM}) - P(t | U, \Theta_{SSM_1})) \prod_{j=2}^n (1 - P(t | U, \Theta_{SSM_j}))$$

for all  $n \leq m$  through induction.

**BC:**  $n = 1$ . As  $A_1 = \frac{A_0 - P(t | U, \Theta_{SSM_1})}{r_1} = \frac{P(t | U, \Theta_{LLM}) - P(t | U, \Theta_{SSM_1})}{r_1}$ , the LHS is  $p(t | u) - P(t | U, \Theta_{SSM_1})$ , which is equal to the RHS.

**IH:** Assume the upper bound holds for all  $n < N$ .

**IS:** Consider the case where  $n = N$ . Then

$$A_N = \frac{A_{N-1} - P(t | U, \Theta_{SSM_N})}{r_N}$$

The LHS is

$$\begin{aligned}
& \frac{A_{N-1} - P(t | U, \Theta_{SSM_N})}{r_N} \prod_{j=1}^N r_j \\
&= (A_{N-1} - P(t | U, \Theta_{SSM_N})) \prod_{j=1}^{N-1} r_j \\
&= \left( A_{N-1} \prod_{j=1}^{N-1} r_j \right) - \left( P(t | U, \Theta_{SSM_N}) \prod_{j=1}^{N-1} r_j \right) \\
&\leq \left( A_{N-1} \prod_{j=1}^{N-1} r_j \right) - P(t | U, \Theta_{SSM_N}) \left( A_{N-1} \prod_{j=1}^{N-1} r_j \right) \\
&\quad (0 \leq A_{N-1} \leq 1 \text{ and all factors non-negative}) \\
&= (1 - P(t | U, \Theta_{SSM_N})) \left( A_{N-1} \prod_{j=1}^{N-1} r_j \right) \\
&\leq (1 - P(t | U, \Theta_{SSM_N})) (P(t | U, \Theta_{LLM}) - P(t | U, \Theta_{SSM_1})) \\
&\quad \prod_{j=1}^{N-1} (1 - P(t | U, \Theta_{SSM_j})) \tag{IH} \\
&= (P(t | U, \Theta_{LLM}) - P(t | U, \Theta_{SSM_1})) \prod_{j=1}^N (1 - P(t | U, \Theta_{SSM_j}))
\end{aligned}$$

which is the RHS, deriving the upper bound as desired.

Thus in all cases,  $P(\text{reject} \cap \mathcal{O}(u) = t | \text{MSS}) \leq \max(0, (P(t | U, \Theta_{LLM}) - P(t | U, \Theta_{SSM_1})) \prod_{i=2}^k (1 - q_i(t)))$ .

We now show  $P(\text{reject} \cap \mathcal{O}(u) = t | \text{NS}) \geq P(\text{reject} \cap \mathcal{O}(u) = t | \text{MSS})$ . Note that  $P(\text{reject} \cap \mathcal{O}(u) = t | \text{NS}) \geq 0$  always, so we only need to compare against  $(P(t | U, \Theta_{LLM}) - P(t | U, \Theta_{SSM_1})) \prod_{j=2}^m (1 - P(t | U, \Theta_{SSM_j}))$  for the cases where  $P(t | U, \Theta_{SSM_k}) < A_{k-1}$  for all  $k$ ; in all other cases  $P(\text{reject} \cap \mathcal{O}(u) = t | \text{MSS}) = 0$  and the inequality is trivially true. In the cases we are interested in,  $P(t | U, \Theta_{SSM_1}) < A_0 = P(t | U, \Theta_{LLM})$ . Then,

$$\begin{aligned}
& P(\text{reject} \cap \mathcal{O}(u) = t | \text{NS}) \\
&= P(t | U, \Theta_{LLM}) \prod_{j=1}^m (1 - P(t | U, \Theta_{SSM_j})) \\
&= (P(t | U, \Theta_{LLM}) - P(t | U, \Theta_{LLM})) P(t | U, \Theta_{SSM_1}) \\
&\quad \prod_{j=2}^m (1 - P(t | U, \Theta_{SSM_j})) \\
&\geq (P(t | U, \Theta_{LLM}) - P(t | U, \Theta_{SSM_1})) \prod_{j=2}^m (1 - P(t | U, \Theta_{SSM_j})) \\
&\quad (P(t | U, \Theta_{LLM}) \leq 1 \text{ and all factors non-negative}) \\
&\geq P(\text{reject} \cap \mathcal{O}(u) = t | \text{MSS})
\end{aligned}$$

Since the inequality holds for all  $t$  in the vocabulary, by the law of total probability, we have  $P(\text{reject} | \text{MSS}) \leq P(\text{reject} | \text{NS})$ .  $\square$

---

# SpQR: A Sparse-Quantized Representation for Near-Lossless LLM Weight Compression

---

**Tim Dettmers<sup>#†</sup>**  
University of Washington

**Ruslan Svirschevski\***  
HSE University & Yandex

**Vage Egiazarian\***  
HSE University & Yandex

**Denis Kuznedelev\***  
Yandex & Skoltech

**Elias Frantar**  
IST Austria

**Saleh Ashkboos**  
ETH Zurich

**Alexander Borzunov**  
HSE University & Yandex

**Torsten Hoefler**  
ETH Zurich

**Dan Alistarh**  
IST Austria & NeuralMagic

## Abstract

Recent advances in large language model (LLM) pretraining have led to high-quality LLMs with impressive abilities. By compressing such LLMs via quantization to 3-4 bits per parameter, they can fit into memory-limited devices such as laptops and mobile phones, enabling personalized use. However, quantization down to 3-4 bits per parameter usually leads to moderate-to-high accuracy losses, especially for smaller models in the 1-10B parameter range, which are well-suited for edge deployments. To address this accuracy issue, we introduce the Sparse-Quantized Representation (SpQR), a new compressed format and quantization technique which enables for the first time *near-lossless* compression of LLMs across model scales, while reaching similar compression levels to previous methods. SpQR works by identifying and isolating *outlier weights*, which cause particularly-large quantization errors, and storing them in higher precision, while compressing all other weights to 3-4 bits, and achieves relative accuracy losses of less than 1% in perplexity for highly-accurate LLaMA and Falcon LLMs. This makes it possible to run 33B parameter LLM on a single 24 GB consumer GPU without any performance degradation at 15% speedup thus making powerful LLMs available to consumer without any downsides. SpQR comes with efficient algorithms for both encoding weights into its format, as well as decoding them efficiently at runtime<sup>3</sup>. Specifically, we provide an efficient GPU inference algorithm for SpQR which yields faster inference than 16-bit baselines at similar accuracy, while enabling memory compression gains of more than 4x.

## 1 Introduction

Pretrained large language models (LLMs) improved rapidly from task-specific performance [WSM<sup>+</sup>18, DCLT19, RWC<sup>+</sup>19], to performing well on general tasks if prompted with instructions [BMR<sup>+</sup>20, WBZ<sup>+</sup>21, Ope23]. While the improved performance can be attributed to scaling in training data and parameters [KMH<sup>+</sup>20, CND<sup>+</sup>22] recent trends focused on smaller models trained on more data, that are easier to use at inference time [HBM<sup>+</sup>22, BSA<sup>+</sup>23, TLI<sup>+</sup>23]. For example, the 7B parameter LLaMA model trained on 1T tokens achieved an average performance only slightly lower than GPT-3 [BMR<sup>+</sup>20] despite being 25x smaller. Current techniques for LLM compression can shrink these models further by a factor of about 4x, while preserving their performance

<sup>\*</sup>Equal contribution

<sup>†</sup>Corresponding author: [dettmers@cs.washington.edu](mailto:dettmers@cs.washington.edu)

<sup>3</sup>[github.com/Vahe1994/SpQR](https://github.com/Vahe1994/SpQR); to be integrated into [github.com/TimDettmers/bitsandbytes](https://github.com/TimDettmers/bitsandbytes)

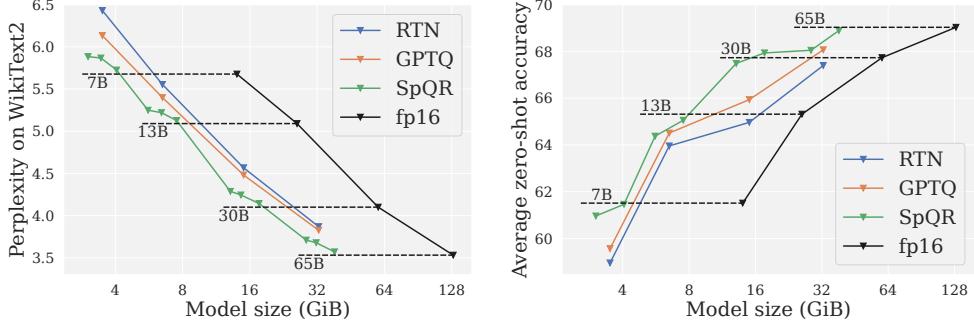


Figure 1: Compressed LLM performance for LLaMA models. **(left)** LM loss on WikiText2 vs model size. **(right)** Average performance on zero-shot tasks vs model size.

[DLBZ22, XLS<sup>+</sup>22, FAHA22, DZ22]. This yields performance levels comparable to the largest GPT-3 model, with major reductions in terms of memory requirements. With such improvements, well-performing models could be efficiently served on end-user devices, such as laptops.

The main challenge is to compress models enough to fit into such devices while also preserving generative quality. Specifically, studies show that, although accurate, existing techniques for 3 to 4-bit quantization still lead to significant accuracy degradation [DZ22, FAHA22]. Since LLM generation is sequential, depending on previously-generated tokens, small relative errors can accumulate and lead to severely corrupted outputs. To ensure reliable quality, it is critical to design low-bitwidth quantization that does not degrade predictive performance compared to the 16-bit model.

In this work, we introduce Sparse-Quantized Representations (SpQR), a hybrid sparse-quantized format which can compress accurate pretrained LLMs to 3-4 bits per parameter while staying *near-lossless*: specifically, SpQR is the first weight quantization method which is able to reach such compression ratios while inducing end-to-end accuracy error as measured in perplexity of less than 1% relative to the dense baseline. SpQR works by combining two innovations. First, we isolate *outlier weights*, whose quantization we show to induce disproportionately high errors: these weights are kept in high precision, while the other weights are stored in a much lower, e.g. 3-bit, format. Second, we implement a variant of grouped quantization with very small group size, e.g. 16 contiguous elements, but we show that one can quantize the quantization scales themselves to a 3-bit representation.

To convert a given pretrained LLM into SpQR format, we adopt an extended version of the post-training quantization (PTQ) approach recently introduced by GPTQ [FAHA22]. Specifically, the method passes calibration data through the uncompressed model; to compress each layer, it applies a layer-wise solver with respect to the L2 error between the outputs of the uncompressed model, and those of the quantized weights. Our approach splits this process into two steps: an “outlier detection” step, in which we isolate weights whose direct quantization has outsize impact on layer output behavior, and an actual compression step, in which most ( $\geq 99\%$ ) of weights are compressed to low-bitwidth, the outliers are extracted, and the whole representation is rendered more efficient by further compressing the quantization metadata.

Our method is motivated by a new analysis showing that LLM weight quantization errors exhibit both vertical and horizontal group correlations, corresponding to systematic large errors corresponding to input feature dimensions and output hidden dimensions. While outlier input features have been observed before [DLBZ22, XLS<sup>+</sup>22], our work is the first to demonstrate that similar outliers occur *in the weights, for particular output hidden dimensions*. Unlike input feature outliers, the output hidden dimension outliers occur only in small segments for a particular output hidden dimension.

Our quantization algorithm isolates such outliers and efficiently encodes a given model in SpQR format. To exploit the resulting structure, we develop a specialized sparse-matrix multiplication algorithm based on the compressed sparse row (CSR) format. To use SpQR for token-by-token generation, we combine this sparse algorithm together with a dense-quantized matrix multiplication for 3-4 bit weights. With this, SpQR reduces the memory footprint of LLMs by a factor of about 3.4x or more without degradation in accuracy, measured as language modeling loss or perplexity, while also being 20-30% faster for LLM generation compared to 16-bit inference.

## 2 Related Work

We focus our discussion on related *post-training quantization (PTQ) methods* [NAVB<sup>+</sup>20], referring the reader to the recent survey of Gholami et al. [GKD<sup>+</sup>21] for full background on quantization. PTQ methods are a popular approach for *one-shot compression* of models with various sizes, based on a limited amount of calibration data, using accurate solvers, usually focused on layer- or group-wise compression sub-problems. Most PTQ methods, such as AdaRound [NAVB<sup>+</sup>20], BitSplit [WCHC20], AdaQuant [HNH<sup>+</sup>21], BRECCQ [LGT<sup>+</sup>21], or OBQ [FSA22] were designed for vision models or small-scale language models, with less than 100M parameters. All these recent approaches tend to use accurate solvers, which would not scale to GPT-scale models in terms of computational or memory cost, as they are 10-1000x larger in size.

Recently, there has been significant interest in obtaining accurate post-training methods that scale to such massive models. Due to computational constraints, early work such as ZeroQuant [YAZ<sup>+</sup>22], LLM.int8() [DLBZ22], and nuQmm [PPK<sup>+</sup>22] used direct rounding of weights to the nearest quantization level, while customizing the quantization granularity (i.e., group size) to trade off space for increased accuracy. LLM.int8() [DLBZ22] suggested isolating “outlier features” which would be quantized separately to higher bit-width. These approaches are able to induce relatively low quantization error, e.g. 5.5% relative LM Loss increase for LLaMA-7B at 4-bit weight quantization, provided that the quantization granularity is low enough. GPTQ [FAHA22] proposed a higher-accuracy approach (e.g., 4% LM Loss increase in the above setting), which works via an approximate large-scale solver for the problem of minimizing the layer-wise squared error.

Dettmers et al. [DZ22] provided an in-depth overview of the accuracy-compression trade-offs underlying these methods, establishing that 4-bit quantization is an optimal point for round-to-nearest-based methods, whereas higher compression can be achieved via data-aware methods such as GPTQ. SparseGPT [FA23] presented an approach to jointly sparsify LLM weights to medium sparsities, together with quantization of the remaining weights to a fixed given bit-width. One common drawback of existing methods is that the accuracy loss relative to the original model is still significant (see Table 1). This is especially relevant to relatively small but easily deployable models, e.g. in the 7-13B parameter range, where existing methods show drastic accuracy drops. We investigate this question here, and provide a new compression format which can lead to near-lossless 3-4 bits compression in this regime.

A related question is that of performing both activation and weight quantization. There is early work [DLBZ22, XLS<sup>+</sup>22, YAZ<sup>+</sup>22], showing that both activations and weights could be quantized to 8-bits with relatively low accuracy impact. These complementary investigations yield interesting insights into the causes of compression error in the case of LLMs. Specifically, [DLBZ22, XLS<sup>+</sup>22] observe the presence of “outlier features” with significantly higher values in the input/output of large LLMs, which induce higher quantization error, and propose different mitigation strategies.

We analyze this phenomenon from the point of view of weight quantization. In particular, we investigate the outlier structure, beyond input feature outliers in the weight matrix. While we find that input feature outliers of the current layer are correlated to hidden unit outliers weight in the previous layer there is not a strict correspondence. Such partially-structured outlier patterns necessitate a fine-grained hybrid compression format that goes beyond algorithms that exploit the column structure of outlier features found in previous work.

Hybrid sparse-quantized formats have been investigated generally for deep networks. Some efficient CPU inference engines [Neu22, GFS<sup>+</sup>19] support a different block sparse-and-quantized format, in which each block of 4 consecutive weights is either completely sparse or quantized to 8-bit format, whereas GPUs support a similar compound format in which every group of 4 weights contains 2 zero weights, and the non-zero weights could be quantized. The FBGEMM package [KHB<sup>+</sup>21] proposed a format in which certain “outlier” weights are quantized separately, to reduce their impact on normalization. However, in this format, “outlier” weights are still quantized to exactly the same bit-width (8-bit) as regular weights; moreover, no procedure is given for converting a model to this format post-training. By contrast, 1) we provide an efficient and accurate post-training compression algorithm which identifies outliers as weights inducing high output error, 2) we propose a format compressing outliers to a higher bit-width relative to regular weights, and 3) our format stores outliers in blocks, allowing for efficient implementation of GPU kernels, which we provide as well.

### 3 Quantization sensitivity of LLM weights

#### 3.1 Parameter sensitivity under quantization

Not all parameters in a neural network are equally important. Intuitively, a weight could be seen as sensitive to quantization if its rounding error is large, i.e. it is not close to a quantization point, and/or the inputs it is usually multiplied with a large, amplifying even a small rounding error. These simple notions of sensitivity however disregard the fact that LLMs operate on very large vectors with significant correlations: a weight  $w_a$  may have a large rounding error while being strongly correlated to another weight  $w_b$ , meaning that the error of rounding up  $w_a$  can be well compensated by rounding down  $w_b$ . This idea is exploited by modern quantization algorithms [FAHA22, YAZ<sup>+</sup>22] and can lead to major improvements over vanilla rounding, especially a low bitwidths. Properly capturing this aspect of sensitivity requires a more robust definition.

For computational tractability, we assess sensitivity on a per-layer level using a small set of *calibration inputs*  $X$ , collected by running them through the model up to the particular layer. We define the sensitivity  $s_{ij}$  of some weight  $w_{ij}$  in the layer’s weight matrix  $W$  as the minimum squared difference between the original predictions on  $X$  and those of any weight matrix  $W'$  where this weight is quantized, i.e.  $w'_{ij} = \text{quant}(w_{ij})$ :

$$s_{ij} = \min_{W'} \|WX - W'X\|_2^2 \quad \text{s.t.} \quad w'_{ij} = \text{quant}(w_{ij}) \quad (1)$$

Crucially, all weights of  $W'$  except for  $w'_{ij}$  may take on arbitrary, not necessarily quantized, values in order to compensate for the quantization error incurred by rounding  $w_{ij}$ , thus capturing the correlation aspect discussed above. Further, as we allow continuous values, this problem admits a closed-form solution. This can be determined by following the generalized Optimal Brain Surgeon framework [FSA22], where  $(XX^\top)^{-1}$  is the inverse Hessian matrix corresponding to the optimization problem:

$$s_{ij} = \frac{(w_{ij} - \text{quant}(w_{ij}))^2}{2(XX^\top)^{-1}}. \quad (2)$$

This saliency measure can be approximated efficiently by quantization solvers, such as GPTQ [FAHA22]. In more detail, GPTQ quantizes weight matrices column-by-column while in each step adjusting the not-yet-quantized part to compensate for the quantization error in a similar sense as defined above. Consequentially, instead of statically deciding all sensitivities in advance, they can be computed dynamically as the algorithm processes each column, by using the inverse of the Hessian subselection corresponding to all not yet quantized weights. This matrix is already efficiently computed by GPTQ and thus does not impose any additional overheads. The main advantage of this approach is that  $s_{ij}$  is always determined based on the most current value of  $w_{ij}$  and thus accounts for adjustments due to previously quantized weights as well.

#### 3.2 Exploring parameter sensitivity

Before we define our main method, SpQR, we provide a motivating analysis of parameter sensitivity which uncovers that the location of sensitive weights in the weight matrix are not random but have particular structures. To highlight these structural elements during the quantization process, we calculate the per-weight sensitivities and visualize them for the popular and highly-accurate LLaMA-65B model [TLI<sup>+</sup>23]. As the quantization method, we use GPTQ quantization to 3-bit, without weight grouping, following [FAHA22]. We use C4 [RSR<sup>+</sup>20] as the calibration dataset, and we estimate the error on 128 sequences of 2048 tokens each. Figure 2 depicts the output projection of the last self-attention layer of LLaMA-65B.

Using the sensitivity analysis, we observe several patterns in the weight matrix, often in a single row or column. Since the large weight matrices in LLaMA-65B have too many rows/columns to be respresentable in a compact image (default: 8k × 32k pixels) we perform max pooling to visualize the matrices, that is we take the maximum sensitivity in each square of 32 × 32 rows and columns. This max pooling only affects the leftmost image. Using this visualization, we observe that the quantization error patterns vary both by layer type, for example attention vs multilayer perceptron (MLP), and layer depth. In particular, we find that more sensitive outliers are present for deeper layers. (Please see Appendix A for additional results.) We now proceed to categorize outlier structures, taking this attention weight matrix as an exemplar. We make the following observations:

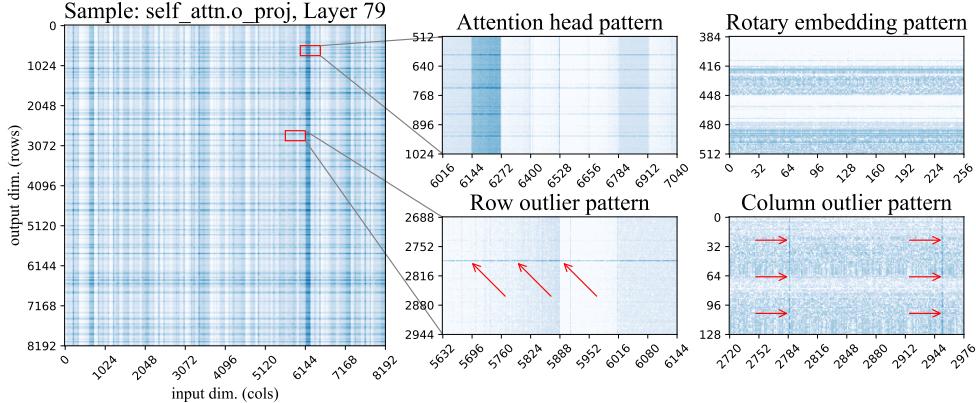


Figure 2: Weight log-sensitivities from the last attention layer of LLaMA-65B. Dark-blue shades indicate higher sensitivity. The image on the left is a high-level view, resized to 1:32 scale with max-pooling. The two images in the middle are zoomed in from the main figure. The two images on the right are taken from other weight matrices.

- **Row outliers** are shown in Figure 2 bottom-center as regions of high sensitivity within one output unit. Some of these patterns span the entire row, while others are partial. In attention layers, some of the partial row outliers correspond to some subset of attention heads. **Column outliers** appear in Figure 2, bottom-right, showing high sensitivity in select input dimensions (columns) across all rows. The latter are correlated to the “outlier feature” phenomenon reported in Dettmers et al. [DLBZ22].
- **Sensitive attention heads.** (Figure 2, top-center) – regular stripes of width 128 highlight all weights corresponding to one attention head. This could be related to some attention heads having more important functions [VTM<sup>+</sup>19, Vig19, OEN<sup>+</sup>22]. The corresponding “stripes” are horizontal for attention Q & K projections, vertical in output projection, and absent from value projections and any MLP weights. Of note, there is significant variation in individual weight sensitivity even within the sensitive heads.
- **The Rotary embedding pattern**, a repeating vertical pattern of sensitivity with a period of 64 units. We attribute this to the use of rotary embeddings [SLP<sup>+</sup>21]: each attention head ( $\text{dim} = 128$ ) is split into two halves: the first 64 are “rotated” with cosine, and the other 64 use sine. Both sine and cosine rotation use the same set of frequencies. Typically, the weights that correspond to low-frequency sines and cosines are more sensitive than their high-frequency counterparts, as shown in Figure 2 (top-right). As expected, this pattern is absent from any layer not using rotary embeddings.
- **Unstructured outliers.** Besides the above, each layer has a number of individual sensitivity weights that do not fit into any of the above patterns. These unstructured outliers occur more frequently for columns with largest input index (i.e. on the right side of the images). This effect is difficult to see on a heatmap, so we provide additional figures and statistical tests in Appendix A. We believe is probably an artefact of the GPTQ algorithm, which compresses one by one, using yet-uncompressed weights to compensate the error. Thus, the rightmost batch of weights accumulates the most error.

Next, we will leverage these findings to propose a compressed representation which can support all these different outlier types.

## 4 SpQR: A Sensitivity-aware compressed representation

### 4.1 Overview

Existing LLM quantization algorithms treat low- and high-sensitivity weights equally; however, our above discussion suggests that this may lead to sub-optimal quantization. Ideally, we would want the representation to assign more of its “size budget” to sensitive weights. However, these weights

are scattered in the weight matrix as either individual weights or small groups, for example, partial rows or attention head. To capture this structure, we are introducing two changes to the quantization procedure: one for capturing small sensitive groups, and another for capturing individual outliers.

**Capturing small groups of weights with bilevel quantization.** In the previous section, we observed several cases where weights behave similarly in small consecutive groups, with abrupt changes between groups, for example for some attention head and partial row outliers (see Figure 4 left, bottom-center). When applying a standard approach, there will be many cases where these weights will be grouped together, sharing the same quantization statistics. To reduce the number of such cases, we use groupwise quantization with extremely small groups, typically of  $\beta_1=8 - 32$  weights. That is, for every  $\beta_1$  consecutive weights, there is a separate quantization scale and zero-point. This choice runs contrary to current intuition: for instance, the recent work of Yao et al. [YLW<sup>+</sup>23] explicitly recommends against small groups, arguing that the overhead for storing quantization statistics would outweigh the precision advantages.

To circumvent this issue, we quantize the groupwise statistics themselves using the same quantization algorithm as for weights — asymmetric (min-max) quantization. Because of how min-max quantization works, the range of quantized values will fit to the groups with largest (or smallest) quantization scale, quantizing them perfectly. In other words, we group groupwise statistics from  $\beta_2 = 16$  consecutive values and quantize them together in the same number of bits, such that groups with atypical quantization parameters end up using more of the “quantization budget”. Finally, both first and second-level quantization is directly within the quantization process, allowing the algorithm to compensate the second-level quantization error where possible.

**High-sensitivity outliers.** Our analysis showed the existence of cases where a small percentage of sensitive weights come in small groups (in the self-attention) or individual “outliers” (in the MLP). In some cases, 1% of the weights account for over 75% of the total quantization error. Since these weights appear to lead to high, irreducible error, we choose to keep these outliers in high precision (16-bit). As these outliers are often unstructured, we encode them individually in a row-wise arrangement similar to a compressed-sparse-row (CSR) representation [HABN<sup>+</sup>21]. This can encode both individual outliers and small structures that do not fit into the above definition of groups.

The procedure for detecting the outliers is described in detail in Alg. 1. It follows a rough two-step procedure: (1) find and isolate outliers as 16-bit weights, (2) quantize the non-outlier “base” weights into 3-4 bit and transfer the remaining quantization into the the 16-bit outliers weights. For the outlier isolation step, the algorithm implements a filtering technique based on the sensitivity criterion in Eq. (2), which is used to isolate and separate outliers from base weights. Globally, for each matrix, the algorithm aims to pick a sensitivity threshold  $\tau$  to obtain the desired number of outliers across the whole model, usually around 1% of weights. Specifically, a particular weight is considered an outlier if keeping the weight in 16-bit reduces the error in Eq. (2) by at least  $\tau$ .

Following this first outlier detection step, we quantize the base weights ignoring all outliers that occur in the same quantization group. As such, the quantization statistics (e.g. scales) are computed by excluding outliers. This results in significant improvements in terms of error, since e.g. the min-max scales will be significantly reduced. The algorithm then proceeds to apply GPTQ to quantize the remaining weights. Interestingly, unlike [DLBZ22], a weight can be chosen to be an outlier not only if it causes error by itself, but also if the GPTQ algorithm can employ this weight to compensate errors from many other weights. Thus, the resulting 16-bit value will contain not the original weight, but a weight that was adjusted to minimize the output error. As such, SpQR goes beyond mere detection of outliers towards the more general notion of isolating and treating outliers that occur *during* the quantization process. Finally, the algorithm gathers and compresses sparse outlier matrix as well as the final quantization statistics with bilevel quantization and returns the compressed weights and their metadata.

**Implementation details.** Our algorithm also contains several optimizations. As we are using small group sizes, it is often the case that a group contains all positive (or all negative) values. Standard quantizers [FSA22, FAHA22] require the maximum value to be positive and the minimum value to be negative. For small group sizes, removing this requirement results in slightly better quality. As a by-product of quantizing the quantization statistics, our algorithm allows non-integer zero points. We ablate these and other SpQR components in Section 5.

**Algorithm 1** SpQR quantization algorithm: the left snippet describes the full procedure, the right side contains subroutines for bilevel quantization and finding outliers.

---

```

func SPQRQUANTIZE( $W, X, b, \beta_1, \beta_2, \tau, \lambda$ )
Input:  $W \in \mathcal{R}^{m \times n}$  — weight matrix,
 $X \in \mathcal{R}^{n \times d}$  — calibration data,
 $b$  — the base number of quantization bits,
 $\beta_1, \beta_2$  — quantization group sizes,
 $\tau$  — sensitivity outlier threshold
 $\lambda$  — hessian regularizer,
1:  $E := \text{float\_matrix}(m, n)$  // L2 error
2:  $H := 2XX^T$  // L2 error hessian,  $\mathcal{R}^{n \times n}$ 
3:  $H^{\text{ic}} := \text{Cholesky}((H + \lambda I)^{-1})$ 
4:  $Q := \text{int\_matrix}(m, n)$  // quantized weight
5:  $\mathcal{O} := \emptyset$  // a set of all outliers
6:  $\mathcal{S} := \emptyset$  // a set of quantization statistics
7: for  $i = 1, \beta_1, 2\beta_1, \dots, n$  do
8:    $W_{:, i:i+\beta_1}, \mathcal{O} := \text{outliers}(W_{:, i:i+\beta_1}, H^{\text{ic}}_{i:(i+\beta_1), i:(i+\beta_1)}, \mathcal{O})$ 
9:    $\hat{s}, \hat{z}, \mathcal{S} := \text{fit\_statistics}(W_{:, i:i+\beta_1}, \mathcal{S}, \mathcal{O})$ 
10:  for  $j = i, \dots, i + \beta_1$  do
11:     $Q_{:, j} := \text{quantize}(W_{:, j}, \hat{s}, \hat{z})$ 
12:     $\vec{w}_q := \text{dequantize}(Q_{:, j}, \hat{s}, \hat{z})$ 
13:     $E_{:, j} := (W_{:, j} - \vec{w}_q) / H^{\text{ic}}_{j,j} \cdot (1 - \text{is\_outlier}(W_{:, j}, \mathcal{O}))$ 
14:     $W_{:, j:(i+\beta_1)} := W_{:, j:(i+\beta_1)} - E \cdot H^{\text{ic}}_{j,j:(i+\beta_1)}$ 
15:     $W_{:, (i+\beta_1):n} := W_{:, (i+\beta_1):n} - E \cdot H^{\text{ic}}_{:, (i+\beta_1), i:(i+\beta_1)}$ 
16:  $S_q, Z_q, S_s, Z_s, S_z, Z_z := \text{gather\_statistics}(\mathcal{S})$ 
17:  $W_{\text{sparse}} = \text{gather\_outlier\_matrix}(W, \mathcal{O})$ 
18: return  $Q, S_q, Z_q, S_s, Z_s, S_z, Z_z, W_{\text{sparse}}$ 

func quantize( $M, \vec{s}, \vec{z}$ )
1: return  $\lfloor M/\vec{s} + \vec{z} + 0.5 \rfloor$ 

func dequantize( $Q, \vec{s}, \vec{z}$ )
1: return  $\vec{s} \cdot (Q - \vec{z})$ 

func fit_quantizer( $M, \beta$ )
1:  $\vec{m} := \text{flatten}(M)$ 
2:  $\vec{s}, \vec{z} := \text{vectors}()$ 
3: for  $i = 1, \beta_1, 2\beta_1, \dots, \dim(m)$  do
4:    $s_i := \frac{\max(\vec{m}_{i:i+\beta}) - \min(\vec{m}_{i:i+\beta})}{2^b - 1}$ 
5:    $z_i := -\min(\vec{m}_{i:i+\beta}) / s_i$ 
6: return  $\vec{s}, \vec{z}$ 

func error( $W, H^{\text{ic}}$ )
1:  $\vec{s}, \vec{z} := \text{fit\_quantizer}(W, \beta_1)$ 
2:  $W_q := \text{quantize}(W, \vec{s}, \vec{z})$ 
3:  $E := (W - W_q) / H^{\text{ic}}$ 
4: return  $E^2$ 

func outliers( $W, H^{\text{ic}}, \mathcal{O}$ )
1:  $E_{\text{base}} = \text{error}(W, H^{\text{ic}})$ 
2: for  $i = 1, \dots, \beta_1$  do
3:    $loo := \{1, 2, \dots, \beta_1\} / \{i\}$ 
4:    $E_{\text{ol}} = \text{error}(W_{:, loo}, H^{\text{ic}}_{loo, loo})$ 
5:    $I_o = \text{select}(E_{\text{base}} - E_{\text{ol}} > \tau)$ 
6:    $\mathcal{O} := \mathcal{O} \cup I_o$ 
7: return  $W, \mathcal{O}$ 

func fit_statistics( $W, \mathcal{S}, \mathcal{O}$ )
1:  $W := W \cdot (1 - \text{is\_outlier}(W, \mathcal{O}))$ 
2:  $\vec{s}, \vec{z} := \text{fit\_quantizer}(W, \beta_1)$ 
3: //  $\vec{s}$  for scales,  $\vec{z}$  for zero points
4:  $\vec{s}_s, \vec{z}_s := \text{fit\_quantizer}(\vec{s}, \beta_2)$ 
5:  $\vec{s}_z, \vec{z}_z := \text{fit\_quantizer}(\vec{z}, \beta_2)$ 
6:  $\vec{s}_q := \text{quantize}(\vec{s}, \vec{s}_s, \vec{z}_s)$ 
7:  $\vec{z}_q := \text{quantize}(\vec{z}, \vec{s}_z, \vec{z}_z)$ 
8:  $\mathcal{S} := \mathcal{S} \cup \{s_q, s_s, s_z, z_q, z_s, z_z\}$ 
9:  $\hat{s} := \text{dequantize}(s_q, s_s, s_z)$ 
10:  $\hat{z} := \text{dequantize}(z_q, z_s, z_z)$ 
11: return  $\hat{s}, \hat{z}, \mathcal{S}$ 

```

---

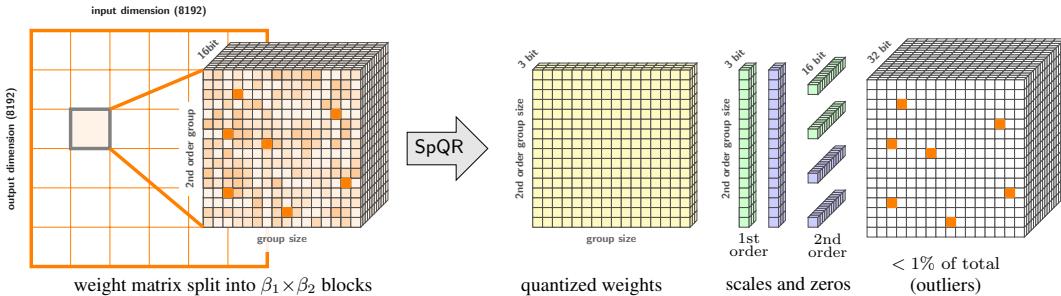


Figure 3: A high-level overview of the SpQR representation for a single weight tensor. The right side of the image depicts all stored data types and their dimensions.

## 4.2 Implementing and Leveraging the Sparse Quantized Representation

Our algorithm converts homogeneous weights into several data structures of various sizes and precisions. Overall, the representation consists of (1) quantized weights, (2) first level quantized quantization statistics, second level quantization statistics, and (3) the CSR outlier indices and values. We summarize the overall structure of SpQR in Figure 3 and describe each component below.

**Storing quantized groups.** All non-outlier weights are encoded as a structure that contains:

- a  $b_w$ -bit individual weight;

- a  $b_q$ -bit scale and zero point for each group of size  $B$ ;
- 16-bit statistics for quantizing groups of  $B_q$  quantization scales and zero-points.

As a particular example for a SpQR representation, consider  $b_w=b_q=3$  and  $B_w=B_q=16$ . The weight matrix is split into groups of  $B_q \times B_w = 256$  weights. A group contains 256 individual  $b_w = 3$ -bit codes. Every 16 weights use a separate 3-bit scale and zero-point. Finally, there are four 16-bit scalars for the entire group used for second level quantization. To simplify GPU memory access, we keep the quantized values for outlier weights in place and adjust the 16-bit versions to compensate for that. We also store both quantized weights and quantized quantization statistics in a contiguous memory region for each group. When running on a different hardware (e.g. mobile CPUs), it is possible to further reduce the memory footprint by removing the quantized version of outliers. We leave this direction for future work.

**Storing outliers.** Recall that our outliers are unstructured; for storage, we sort them by their row first and column second, so that outliers in the same row are contiguous in memory. For each outlier, we store two scalars: the 16-bit weight value and the 16-bit column index. For each row, we also store a single 32-bit number—the total number of outliers in the rows up to the current one for efficient inference. This results in an average storage cost of 32.03 to 32.1 bits per sensitive weight. This could be reduced significantly by grouping outliers, which we leave as future work.

**Inference with SpQR.** To illustrate the practicality of our approach, we design an efficient GPU-based decoding implementation for the SpQR format, focused on the popular token-by-token LLM generation as a use-case.

We leverage the fact that autoregressive inference on GPUs is memory-bound, so high compression rates can hide decoding overheads, to a significant extent. At a high level, our algorithm loads group statistics and the quantized weights into shared memory (SRAM), dequantizes to 16-bits, and then performs matrix multiplication with 16-bit inputs. For handling outliers, we design a sparse matrix algorithm that takes advantage of outliers that occur in rows. Roughly, the algorithm works as follows

First, (1) we divide the matrix into equally sized blocks. Then, each GPU core (thread block) (2) loads a large slice of outliers into shared memory (SRAM), and each GPU core (3) determines if outliers are part of the segment or not. The corresponding weights are (4) loaded from main memory; finally, the matrix multiplication is performed.

This algorithm essentially performs load balancing through steps (1-3), while step (4) tends to have contiguous memory access due to the row-like patterns for the outliers. We will show in Section 5 that this custom approach is faster than the sparse matrix algorithms in PyTorch.

## 5 Experimental Validation

**Experimental setup.** We focus on three main settings: 1) evaluating what is the most compact representation with which SpQR can replicate the performance of a 16-bit model within 1% perplexity, 2) controlling for the average number of bits per parameter across methods and assess the performance of SpQR compared to round-to-nearest and GPTQ baselines, 3) what is the best trade-off in terms of model size and performance. For these settings, we evaluate the full SpQR algorithm on publicly-available LLMs. We focus on the LLaMA {7, 13, 30, 65}B model family [TLI<sup>+</sup>23] and Falcon{7, 40}B model family [UAE23a]. We quantize LLaMa models using the RedPajama dataset and Falcon models on RefinedWeb dataset [UAE23b], publicly-available replicas of the LLaMA and Falcon training data, respectively. In addition, we provide perplexity results for OPT models in Appendix F.

We compare SpQR against two other post-training quantization schemes: GPTQ [FAHA22] and simple rounding-to-nearest (RTN) quantization, which is used by most other LLM compression methods [DLBZ22, YAZ<sup>+</sup>22]. Both baselines use 4-bit quantization since it provides the best quality to size trade-off [DZ22]. For SpQR, we consider both 3-bit and 4-bit base quantization, though the resulting model size can be slightly larger due to the presence of outliers.

We evaluate quantized model performance by two metrics. Firstly, we measure *perplexity*, measured on the WikiText2 [MXBS16], Penn Treebank [MKM<sup>+</sup>94] and C4 [RSR<sup>+</sup>20] datasets. Secondly, we measure zero-shot accuracy on five tasks: WinoGrande [SBBC21], PiQA [TP03], HellaSwag, ARC-easy and ARC-challenge [CCE<sup>+</sup>18]. We use the LM Evaluation Harness [GTB<sup>+</sup>21] with

LLaMa

LLaMa						
Size	Method	Avg bits	Wiki2	C4	PTB	
7B	–	16.00	5.68	7.08	8.80	
	SpQR	4.63	5.73	7.13	8.88	
	RTN	4	6.43	7.93	10.30	
	GPTQ	4	6.13	7.43	9.27	
	SpQR	3.94	5.87	7.28	9.07	
	–	16.00	5.09	6.61	8.07	
13B	SpQR	4.63	5.13	6.64	8.13	
	RTN	4	5.55	6.98	8.65	
	GPTQ	4	5.40	6.84	8.44	
	SpQR	3.96	5.22	6.72	8.22	
	–	16.00	3.53	5.62	6.91	
	SpQR	4.71	3.57	5.64	6.93	
65B	RTN	4	3.87	5.85	7.17	
	GPTQ	4	3.83	5.80	7.07	
	SpQR	3.90	3.68	5.70	6.99	
	–	16.00	3.53	5.62	6.91	
	SpQR	4.71	3.57	5.64	6.93	
	RTN	4	3.87	5.85	7.17	

Table 1: Perplexity on WikiText2 [MXBS16], C4 [RSR<sup>+</sup>20] and Penn Treebank [MKM<sup>+</sup>94] for SpQR and round-to-nearest (RTN) and GPTQ baselines with LLaMa. We can see that SpQR reaches performances within 1% of the perplexity with less than 4.71 bits per parameter. We also see that for 4-bits per parameter SpQR significantly improves on GPTQ with an improvement as large as the improvement from RTN to GPTQ.

recommended parameters. We provide full configurations in Appendix B, as well as code which we plan to release publicly. Our implementation takes around 4.5 hours on the largest model size (65B) on an NVIDIA A100 and about 6 on an A6000.

To control for model size, we evaluate RTN and GPTQ with 4-bit base quantization. For SpQR we use 3-bit base quantization, a group size of 8 with 3-bit for the first quantization, a group size of 64 for the second quantization, and as many outliers as possible to still reach less than 4-bits per parameter on average. We aim to achieve *near-lossless* compression, for which we adopt the definition of the MLCmons benchmark [RCK<sup>+</sup>20]: 1% error relative to the uncompressed baseline. In all SpQR evaluations, we choose  $\tau$  such that the proportion of outliers is under 1%.

**Main Results.** Figure 1 measures actual model size versus perplexity on LLaMa models on WikiText2, and accuracy on zero-shot tasks. We observe that SpQR outperforms GPTQ (and correspondingly RTN) at similar model size by a significant margin, especially on smaller models. This improvement comes from both SpQR achieving more compression, while also reducing loss degradation. In addition, if we measure the bits per parameter needed to come within 1% of the 16-bit performance in terms of perplexity, Figure 1 shows that SpQR with 4.6 to 4.71 bits per parameter approaches the non-quantized models with at most 1% margin of error for all models (see Table 1 and Table 2 for exact values).

The second set of results, presented in Table 1 for LLaMa and Table 2 for Falcon family models, controls model size by comparing SpQR and baseline methods with 4 bits per parameter. These results show that SpQR improves over previous methods, with the gap between SpQR and the next best method GPTQ being as large as the improvement of GPTQ over naive RTN. For 4-bit, SpQR halves the error relative to the 16-bit baseline compared to GPTQ.

**Ablations.** The SpQR representation differs from standard quantization methods in two main ways: bilevel quantization with small quantization group size and unstructured outliers. To understand the effect of small group sizes, we compare 3-bit SpQR with group size 16, compressed using 3-bit bilevel quantization, versus a setup with group size 48, keeping quantization statistics in 16-bit. Both configurations result in approximately 3.6 average bits per parameter. For simplicity, neither uses outliers. We report both in Table 3, the “3-bit statistics” entry corresponds to group size 16 with 3-bit statistics and “16-bit statistics” stands for group size 16 with 16-bit statistics. Given the same (slightly smaller) memory footprint, using quantized statistics significantly improves language modeling loss.

Next, we ask whether it is necessary to use unstructured outliers, considering two outlier types. First, we use the criterion of Dettmers et al. [DZ22] to find column outliers and quantize them in higher precision. The alternative is to treat the entire rows (output units / hidden units / neurons) as outliers: we run SpQR without outliers, then select  $k$  output units that have the highest quantization error (i.e.

### Falcon

Size	Method	Avg bits	Wiki2	C4	PTB	Size	Method	Avg bits	Wiki2	C4	PTB
7B	—	16.00	6.59	9.50	9.90	40B	—	16.00	5.23	7.76	7.83
	SpQR	4.44	6.64	9.58	9.97		SpQR	4.46	5.26	7.79	7.86
	RTN	4	8.73	12.56	13.76		RTN	4	6.52	9.76	10.63
	GPTQ	4	6.91	9.93	10.33		GPTQ	4	5.36	7.95	8.01
	SpQR	3.92	6.74	9.70	19.114		SpQR	3.90	5.29	7.85	7.91

Table 2: Perplexity on WikiText2 [MXBS16], C4 [RSR<sup>+</sup>20] and Penn Treebank [MKM<sup>+</sup>94] for SpQR and round-to-nearest (RTN) and GPTQ baselines on Falcon model. We can see that SpQR reaches performances within 1% of the perplexity with less than 4.5 bits per parameter. We also see that for 4-bits per parameter SpQR significantly improves on GPTQ with an improvement as large as the improvement from RTN to GPTQ.

Name	Wiki2	C4	PTB	Avg bits
Uncompressed	3.53	5.62	6.91	16
GPTQ (4 bit)	3.83	5.80	7.07	4
3-bit statistics	3.74	5.73	7.02	3.63
16-bit statistics	3.84	5.83	7.12	3.67
Round zero	3.75	5.76	7.01	3.63
w/o act order	3.74	5.76	7.05	3.63

Table 3: Perplexity for LLaMA-65B model.

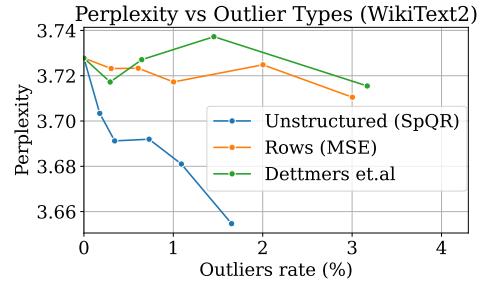


Figure 4: Different outlier types, LLaMA-65B.

MSE between layer predictions) and treat the entire rows as 16-bit outliers. We compare the three outlier types on top of 3-bit SpQR and report the results in Figure 4. Overall, unstructured outliers reduce perplexity significantly faster than their row counterpart and the criterion of [DZ22], even after accounting for the different memory footprint.

Finally, we analyze the impact of the minor hyperparameter changes that we introduced at the end of Section 4. In Table 3 (bottom), we evaluate quantization errors without these changes. The “Round zero” entry corresponds to a version of SpQR where the zero-point is a 3-bit integer. This reduces the memory footprint of SpQR, but results in a moderate increase in perplexity. Similarly, we evaluate SpQR without the “act order” flag. This option re-orders the input dimensions by the diagonal of the inverse hessian, which was introduced as a part of the GPTQ algorithm. Using this heuristic slightly improves loss, though not as much as from quantized groups.

To summarize, both small quantized groups and unstructured outliers independently improve perplexity and perform better than alternative strategies. SpQR also benefits from using the GPTQ activation order heuristic, though the gain is smaller than from outliers or small groups. Still, we opt to use the same activation order heuristic in the GPTQ baselines to ensure a fair comparison. To further explore the design space of SpQR, we provide an additional hyperparameter study in Appendix C.

**Inference Time.** Finally, we evaluate the inference speed of SpQR for autoregressive inference with a focus on measuring the token generation latency with batch size 1 on a single A100 GPU. We measure inference speed in two setups: i) generating 100 tokens from scratch and ii) adding 100 tokens on top of a 1024-token prefix (prompt). We compare our specialized sparse matrix multiplication algorithm with the algorithm implemented in PyTorch (cuSPARSE). We also compare against a 16-bit baseline. We measure the end-to-end latency as inference steps per second for the full SpQR algorithm, that is for both the dense and sparse multiplication part together.

Results are shown in Table 4. We can see that while standard sparse matrix multiplication in PyTorch is not faster than 16-bit inference, our specialized sparse matrix multiplication algorithm yields speedups of about 20-30%.

Method	fp16 (baseline)				SpQR (PyTorch)				SpQR (optimized)			
LLaMA	7B	13B	30B	65B	7B	13B	30B	65B	7B	13B	30B	65B
scratch	$47 \pm 2.3$	$37 \pm 0.8$	$19 \pm 1.1$	OOM	$30 \pm 2.2$	$24 \pm 1.2$	$8.8 \pm 0.4$	OOM	$57 \pm 2.4$	$44 \pm 0.5$	$22 \pm 0.9$	$12 \pm 0.6$
prefix 1024	$46 \pm 2.4$	$31 \pm 0.9$	$17 \pm 0.8$	OOM	$27 \pm 1.6$	$21 \pm 1.1$	$6.5 \pm 0.7$	OOM	$55 \pm 2.1$	$37 \pm 0.8$	$22 \pm 1.3$	$11 \pm 0.6$

Table 4: Inference speed comparison (tokens/s), OOM means the model did not fit in an A100 GPU. We see that our optimized SpQR algorithm is faster than the 16-bit baseline and almost 2.0x faster than quantized matrix multiplication + standard PyTorch sparse matrix multiplication baseline.

## 6 Discussion & Limitations

We have presented SpQR, an quantization approach which quantizes sensitive outliers in higher precision, to achieve near-lossless 16-bit accuracy with less than 4.75 bits per parameter on average. We achieve even better quality-size-tradeoff when compressing to as little as 3.36 bits which makes SpQR an ideal method for compressing models for memory-limited devices. Despite our promising results, there are several limitations. The main limitation is that we do not evaluate the generative quality of quantized LLMs, but only the predictive performance in terms of zero-shot accuracy and perplexity. While we believe that perplexity measurements and generation quality are strongly related, this is a hypothesis we aim to investigate in future work. While we devise a sparse matrix multiplication algorithm to accelerate the computation with outliers, another limitation is that we do not fuse sparse matrix multiplication with regular quantized matrix multiplication. Such an approach would yield even better inference time performance. However, such an approach is also very difficult to implement. We leave the implementation of such an algorithm to future work.

## 7 Acknowledgements

D.K. was supported by Russian Science Foundation, grant 21-11-00373. D.A. and E.F. gratefully acknowledge funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 805223 ScaleML). Authors also thank Ivan Komarov for his help in profiling and understanding the performance bottlenecks of SpQR on GPU.

## References

- [BMR<sup>+</sup>20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.
- [BSA<sup>+</sup>23] Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O’Brien, Eric Hallahan, Mohammad Aftab Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. *arXiv preprint arXiv:2304.01373*, 2023.
- [CCE<sup>+</sup>18] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.
- [CND<sup>+</sup>22] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.
- [DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019.
- [DLBZ22] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022*, 2022.

- [DZ22] Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. *arXiv preprint arXiv:2212.09720*, 2022.
- [FA23] Elias Frantar and Dan Alistarh. Massive language models can be accurately pruned in one-shot. *arXiv preprint arXiv:2301.00774*, 2023.
- [FAHA22] Elias Frantar, Saleh Ashkboos, Torsten Hoefer, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.
- [FSA22] Elias Frantar, Sidak Pal Singh, and Dan Alistarh. Optimal Brain Compression: A framework for accurate post-training quantization and pruning. *arXiv preprint arXiv:2208.11580*, 2022. Accepted to NeurIPS 2022, to appear.
- [GFS<sup>+</sup>19] Yury Gorbachev, Mikhail Fedorov, Iliya Slavutin, Artyom Tugarev, Marat Fatekhov, and Yaroslav Tarkan. Openvino deep learning workbench: Comprehensive analysis and tuning of neural networks inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019.
- [GKD<sup>+</sup>21] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.
- [GTB<sup>+</sup>21] Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, Jason Phang, Laria Reynolds, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, September 2021.
- [HABN<sup>+</sup>21] Torsten Hoefer, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *arXiv preprint arXiv:2102.00554*, 2021.
- [HBM<sup>+</sup>22] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.
- [HNH<sup>+</sup>21] Itay Hubara, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry. Accurate post training quantization with small calibration sets. In *International Conference on Machine Learning (ICML)*, 2021.
- [KHB<sup>+</sup>21] Daya Khudia, Jianyu Huang, Protonu Basu, Summer Deng, Haixin Liu, Jongsoo Park, and Mikhail Smelyanskiy. Fbgemm: Enabling high-performance low-precision deep learning inference. *arXiv preprint arXiv:2101.05615*, 2021.
- [KMH<sup>+</sup>20] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.
- [LGT<sup>+</sup>21] Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. BRECQ: Pushing the limit of post-training quantization by block reconstruction. In *International Conference on Learning Representations (ICLR)*, 2021.
- [MKM<sup>+</sup>94] Mitch Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. The penn treebank: Annotating predicate argument structure. In *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*, 1994.
- [MXBS16] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.
- [NAVB<sup>+</sup>20] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or down? Adaptive rounding for post-training quantization. In *International Conference on Machine Learning (ICML)*, 2020.

- [Neu22] NeuralMagic. DeepSparse, 2022.
- [OEN<sup>+</sup>22] Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022.
- [Ope23] OpenAI. Gpt-4 technical report. *arXiv*, 2023.
- [PGM<sup>+</sup>19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Conference on Neural Information Processing Systems (NeurIPS)*. 2019.
- [PPK<sup>+</sup>22] Gunho Park, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. nuQmm: Quantized matmul for efficient inference of large-scale generative language models. *arXiv preprint arXiv:2206.09557*, 2022.
- [RCK<sup>+</sup>20] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459. IEEE, 2020.
- [RSR<sup>+</sup>20] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.
- [RWC<sup>+</sup>19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- [SBBC21] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: an adversarial winograd schema challenge at scale. *Commun. ACM*, 64(9):99–106, 2021.
- [SLP<sup>+</sup>21] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.
- [TLI<sup>+</sup>23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- [TP03] Sandeep Tata and Jignesh M Patel. PiQA: An algebra for querying protein data sets. In *International Conference on Scientific and Statistical Database Management*, 2003.
- [UAE23a] TII UAE. The falcon family of large language models. <https://huggingface.co/tiuae/falcon-40b>, May 2023.
- [UAE23b] TII UAE. The refined web dataset. <https://huggingface.co/datasets/tiuae/falcon-refinedweb>, May 2023.
- [Vig19] Jesse Vig. A multiscale visualization of attention in the transformer model. *arXiv preprint arXiv:1906.05714*, 2019.
- [VTM<sup>+</sup>19] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5797–5808, Florence, Italy, July 2019. Association for Computational Linguistics.

- [WBZ<sup>+</sup>21] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.
- [WCHC20] Peisong Wang, Qiang Chen, Xiangyu He, and Jian Cheng. Towards accurate post-training network quantization via bit-split and stitching. In *International Conference on Machine Learning (ICML)*, 2020.
- [WMR<sup>+</sup>21] Laura Weidinger, John Mellor, Maribeth Rauh, Conor Griffin, Jonathan Uesato, Po-Sen Huang, Myra Cheng, Mia Glaese, Borja Balle, Atoosa Kasirzadeh, Zac Kenton, Sasha Brown, Will Hawkins, Tom Stepleton, Courtney Biles, Abeba Birhane, Julia Haas, Laura Rimell, Lisa Anne Hendricks, William Isaac, Sean Legassick, Geoffrey Irving, and Iason Gabriel. Ethical and social risks of harm from language models, 2021.
- [WSM<sup>+</sup>18] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.
- [XLS<sup>+</sup>22] Guangxuan Xiao, Ji Lin, Mickael Seznec, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*, 2022.
- [YAZ<sup>+</sup>22] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *arXiv preprint arXiv:2206.01861*, 2022.
- [YLW<sup>+</sup>23] Zhewei Yao, Cheng Li, Xiaoxia Wu, Stephen Youn, and Yuxiong He. A comprehensive study on post-training quantization for large language models, 2023.

## Table of contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Related Work</b>	<b>3</b>
<b>3</b>	<b>Quantization sensitivity of LLM weights</b>	<b>4</b>
3.1	Parameter sensitivity under quantization . . . . .	4
3.2	Exploring parameter sensitivity . . . . .	4
<b>4</b>	<b>SpQR: A Sensitivity-aware compressed representation</b>	<b>5</b>
4.1	Overview . . . . .	5
4.2	Implementing and Leveraging the Sparse Quantized Representation . . . . .	7
<b>5</b>	<b>Experimental Validation</b>	<b>8</b>
<b>6</b>	<b>Discussion &amp; Limitations</b>	<b>11</b>
<b>7</b>	<b>Acknowledgements</b>	<b>11</b>
<b>A</b>	<b>Additional weight sensitivity analysis</b>	<b>15</b>
<b>B</b>	<b>Experimental Configurations</b>	<b>18</b>
<b>C</b>	<b>Hyperparameter sensitivity</b>	<b>18</b>
<b>D</b>	<b>Estimating model size</b>	<b>18</b>
<b>E</b>	<b>Choice of optimal configuration for fixed average number of bits</b>	<b>19</b>
<b>F</b>	<b>Additional results for near-lossless compression</b>	<b>20</b>
<b>G</b>	<b>Choice of optimal LLM configuration for specific hardware</b>	<b>20</b>
<b>H</b>	<b>Sensitivity to random seed</b>	<b>22</b>
<b>I</b>	<b>Generative examples</b>	<b>22</b>
<b>J</b>	<b>Broader impact</b>	<b>22</b>
<b>K</b>	<b>On the use of LLMs in this work</b>	<b>26</b>

## A Additional weight sensitivity analysis

In this section, we provide additional visualizations of LLaMA weight sensitivities, as well as additional plots for different layer roles. As we observed earlier in Section 3.2, the sensitivity matrices vary based on four main factors:

- the quantization scheme (e.g. row- or group-wise);
- the layer depth, i.e. the index of the corresponding transformer block;
- the role of that weight, e.g. self-attn query / key or MLP up / down projection;
- the location within the chosen weight matrix;

Here, we report additional observations about these factors and elaborate on some of our claims from Section 3.1. We also report raw sensitivity matrices for various weight matrices at the end of the supplementary materials.

**Relation between sensitivity and the chosen quantization scheme.** We compare two configurations of GPTQ 3-bit. The first configuration uses one quantization scale & zero for each row. The second one uses blockwise quantization with one set of statistics for each block of 128 weights.

Figure 5 demonstrates a typical example of how group size affects sensitivity. In the bottom-right plot, we observe that a subset of weights (width 128) has a significantly higher quantization error

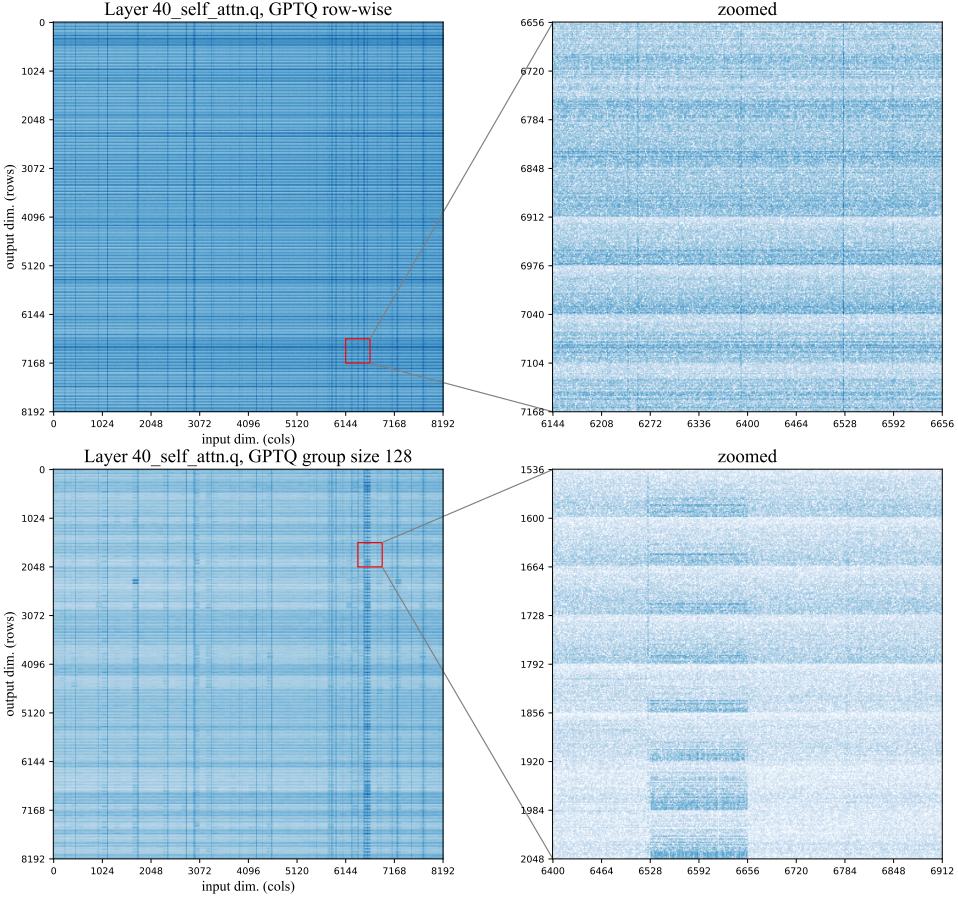


Figure 5: The weight sensitivities for LLaMA-65B 40th layer, attention query projection. The color scale represents sensitivity on a logarithmic scale, with higher sensitivity being darker. **(top)** 3-bit GPTQ with per-row quantization scales, **(bottom)** 3-bit GPTQ with block size 128.

than the rest of the layer. Please note that the color scale represents sensitivity on a logarithmic scale, with higher sensitivity being darker.

On a more detailed examination, we found that this specific group contains a “vertical” outlier, i.e. the corresponding input feature has significantly higher variance, compared to other input dimensions.

In this example, the main effect of GPTQ block size 128 is that the problematic dimension leads to increased sensitivity in a group of  $8192 \times 128$  weights. In turn, GPTQ with per-row statistics has high quantization error across the entire row.

**The effect of rotary embeddings.** Earlier in Figure 2 we note that attention query and key have a regular pattern of sensitivity that repeats every 64 rows. We attribute this to the fact that LLaMA uses rotary position embeddings. More specifically, this pattern is likely a side-effect of how rotary embeddings are implemented for this model.

To recall, rotary position embeddings are a technique that rotates attention head dimensions by an angle that depends on how many tokens are between key and query [SLP<sup>+</sup>21]. Furthermore, dimensions within each head are rotated with a different frequency. To implement this rotation, LLaMA multiplies each head by a precomputed tensor of sine and cosine functions with a different period. The first half (64 units) of the matrix is multiplied by cosines and the other half (64 units) is multiplied by sines.

To recall, sine and cosine components are equivalent up to a phase shift and show similar behavior in our analysis. In general, we observe that weights that correspond to low-frequency heads (bottom of each semi-head) typically have higher sensitivity. One possible explanation is that high-frequency

heads can be more dependent on position-specific information, such as attending to the previous token — and less dependent on the weights that represent content information. However, this phenomenon merits further investigation and our current understanding should be treated as an educated guess.

**GPTQ and the effect of quantization order.** As we observe earlier in Section 3.2, the rightmost weights in each visualization tend to have higher quantization errors. This is likely a side-effect of the GPTQ algorithm, which compresses weights one input feature at a time, i.e. column by column in a left-to-right direction. Once a column is quantized, the algorithm uses the remaining unquantized weights to compensate for the error. Thus, the rightmost batch of weights accumulates the most error from preceding columns and has the least space to compensate it’s “own” quantization error.

This difference is most pronounced in the earlier layers, where the quantization error is smaller overall (see Figure 6). To further verify this observation, we observe that this effect disappears if we shuffle the weight quantization order in the GPTQ algorithm.

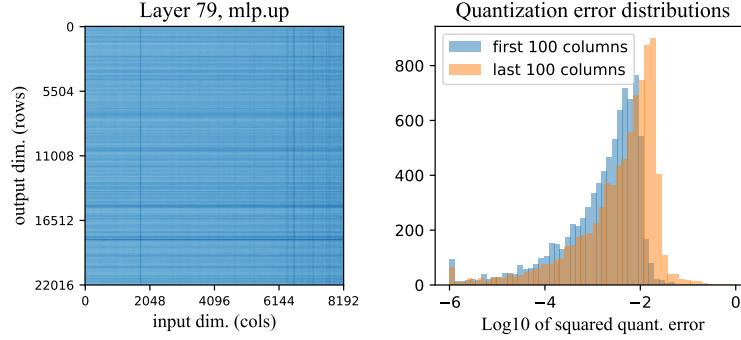


Figure 6: The weight log-sensitivities for a deeper upward projection layer (in particular, this is layer #79). The heatmap on the left represents the sensitivities of each weight, with darker being more sensitive; the histogram on the right captures the sensitivities in the first 100 and last 100 columns (sorted across input dimensions). The latter figure clearly shows that later columns are more sensitive on average.

**Relation between weight sensitivity and layer depth.** In terms of mean squared error, we observe that the first layers of LLaMA tend to have generally lower OBC error (defined as L2 distance between original and quantized layer predictions). To illustrate this, we report the average quantization error of GPTQ-3bit in Figure 7.

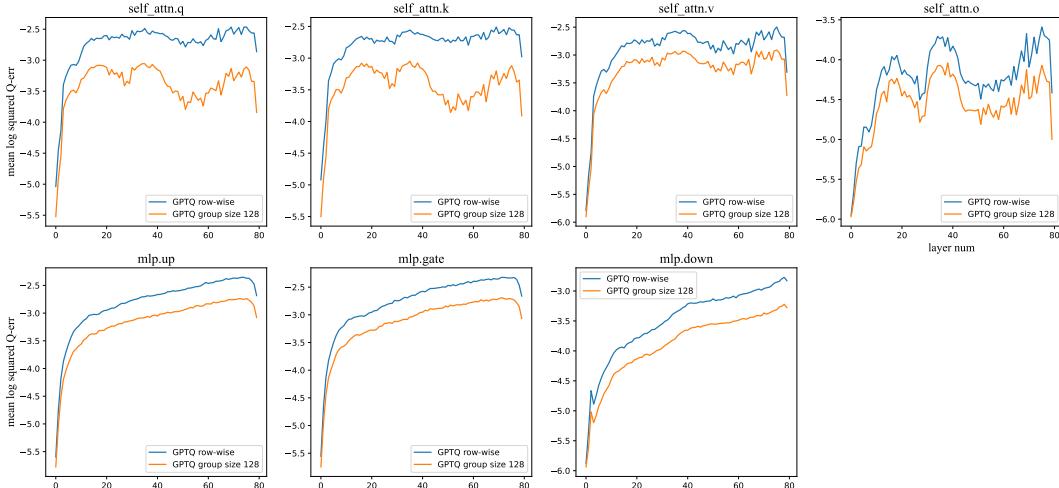


Figure 7: Figure: mean quantization error (vertical axis) as a function of layer depth (horizontal axis). Each plot corresponds to a different layer role.

The absolute quantization error means little by itself since each quantized layer has a different input/output variance. However, we also observe that the first and last few layers have qualitative differences in behavior. Figures 10 and 11 report weight sensitivities for the first, middle (40th), and last (79th) layer of LLaMA model separately to better illustrate this difference.

## B Experimental Configurations

The SpQR representations proposed in this work have several adjustable hyperparameters that allow for great flexibility in targeting a desired size of the model. We introduce the notation and list the method hyperparameters below:

- $b_w$  - number of bits per weight
- $b_s$  - number of bits per scale
- $b_z$  - number of bits per zero
- $r_o$  - outlier rate (fraction of weights that are not quantized)
- $\beta_1$  - block size for weight quantization
- $\beta_2$  - block size for statistic quantization;
- $\tau$  - outlier threshold

The actual number of outliers depends not only on  $\tau$ , but on all other hyperparameters as well. However, for any specific configuration, increasing  $\tau$  leads to reduced number of outliers. To achieve the desired number of outliers, we tune  $\tau$  in [0.1, 1.0] range by binary search with minimum step size 0.05. The vast majority of our configurations are between  $\tau = 0.1$  and  $\tau = 0.45$ .

The full configuration we use to compress LLaMA-30B model near-losslessly in Table 1 has the following hyperparameters:  $b_w = 4, b_s = b_z = 3, \beta_1 = \beta_2 = 16, \tau = 0.1$ . This translates to the following command line arguments in our supplementary code:

```
python main.py $MODEL custom --custom_data_path=$DATA \
--wbits 4 --groupsize 16 --perchannel --qq_scale_bits 3 \
--qq_zero_bits 3 --qq_groupsize 16 --outlier_threshold 0.1 \
--fit_quantizer_without_outliers --permutation_order act_order
```

## C Hyperparameter sensitivity

In this section, we analyze how SpQR performance depends on the choice of quantization group sizes. Please recall that the SpQR algorithm uses two types of groups, indexed by parameters  $\beta_1$  and  $\beta_2$ . The first group dimension  $\beta_1$  covers multiple weights for the same input unit, similar to standard blockwise quantization. In turn, the other dimension  $\beta_2$  covers multiple output units, and is used when quantizing quantization scales. In our visualizations,  $\beta_1$  blocks are always horizontal, while  $\beta_2$  are vertical.

In Table 5, we evaluate SpQR with varying parameters  $\beta_1$  and  $\beta_2$ . We quantize LLaMA-65B with 3-bit SpQR for weights and statistics and report perplexity on WikiText2, Penn Treebank, and C4 datasets. The upper-left section of the table contains the effective number of bits for each group configuration, and the remaining sections correspond to perplexities on different datasets.

## D Estimating model size

In this section, we provide a quick way to estimate the compressed model size before running the quantization. We express this estimate in terms of *average bits per parameter* defined as:

$$\bar{b} = \frac{\text{model size in bits}}{\text{number of parameters}} \quad (3)$$

Where model size in bits denotes the total amount of memory - the quantized weights, 1st-order and 2nd-order quantization statistics, outliers and the outlier index - required for the storage of the model. According to Section 4.2, each outlier requires memory storage of  $\sim 32$  bits.

		Average bits						Wikitext2 Perplexity (3.53)					
$\beta_1 \backslash \beta_2$		4	8	16	32	64	128	4	8	16	32	64	128
4	4	8.5	6.5	5.5	5	4.75	4.625	3.581	3.628	3.715	3.822	4.003	4.23
8	8	5.75	4.75	4.25	4	3.875	3.813	3.625	3.64	3.649	3.666	3.688	3.713
16	16	4.375	3.875	3.625	3.5	3.438	3.406	3.701	3.71	3.728	3.726	3.739	3.741
32	32	3.688	3.438	3.313	3.25	3.219	3.203	3.803	3.797	3.812	3.812	3.815	3.85
64	64	3.344	3.219	3.156	3.125	3.109	3.102	3.884	3.901	3.907	3.899	3.928	3.926
128	128	3.172	3.109	3.078	3.063	3.055	3.051	3.982	3.994	4.005	3.992	4.017	4.013
		C4 Perplexity (5.62)						PTB Perplexity (6.91)					
$\beta_1 \backslash \beta_2$		4	8	16	32	64	128	4	8	16	32	64	128
4	4	5.652	5.674	5.718	5.796	5.919	6.119	6.934	6.965	7.001	7.054	7.194	7.395
8	8	5.683	5.688	5.696	5.703	5.709	5.718	6.962	6.98	6.991	6.99	6.979	7.029
16	16	5.735	5.735	5.735	5.738	5.741	5.749	7.018	7.013	7.015	7.016	7.012	7.03
32	32	5.793	5.789	5.792	5.796	5.794	5.802	7.042	7.053	7.083	7.043	7.069	7.083
64	64	5.857	5.859	5.858	5.866	5.863	5.866	7.084	7.129	7.137	7.118	7.137	7.12
128	128	5.932	5.931	5.935	5.939	5.944	5.936	7.185	7.197	7.232	7.234	7.217	7.199

Table 5: Weight block size  $\beta_1$  and statistic block size  $\beta_2$  performance on WikiText2, C4, and Penn Treebank (PTB). The uncompressed baseline value is provided in the corresponding heading.

The storage and computational cost in transformer models are dominated by the linear projections in the attention and feedforward blocks. Consider quantization of a weight matrix (any of these)  $\mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$  with input dimension  $d_{\text{in}}$  and output dimension  $d_{\text{out}}$ . Then the average number of bits for a given configuration is:

$$\bar{b} \simeq \frac{b_w d_{\text{out}} d_{\text{in}} + (b_s + b_z) \frac{d_{\text{out}} d_{\text{in}}}{\beta_1} + 2(16 + 16) \frac{d_{\text{out}} d_{\text{in}}}{\beta_1 \beta_2} + 32 r_o}{d_{\text{out}} d_{\text{in}}} = b_w + \frac{b_s + b_z}{\beta_1} + \frac{64}{\beta_1 \beta_2} + 32 r_o \quad (4)$$

Therefore, to increase (decrease) the size of the model one should either increase (decrease) the precision of model weights and quantization statistics or decrease (increase) the block size.

For example, for configuration with  $b_w = 3, b_s = 3, b_z = 3, \beta_1 = 16, \beta_2 = 32$  and 0.4% of outliers, the average number of bits is:

$$3 + \frac{3+3}{16} + \frac{64}{16 \cdot 32} + 0.004 \cdot 32 \simeq 3.63$$

## E Choice of optimal configuration for fixed average number of bits

As discussed above our method has multiple options for improvement of model performance at the cost of the increase of the model size: number of bits per weight  $w_b$ , groupsizes  $b_1$  and  $b_2$  for 1st and 2nd order quantization and the outlier rate. We evaluated several configurations with various options for the aforementioned parameters on perplexity benchmarks. Results are presented on Figure 8. One can observe that small groups and small fraction of outliers allows to considerably improve model performance, but the gain is diminishing with the number of bits added (when the additional budget from small group is of order 0.1-0.5 of bits per parameter). It is better to store weights in higher precision instead of keeping them in lower precision but with very small groups or keeping large fraction of outliers. In our experiments optimal fraction of outliers is 0.2-0.5% depending on the model and groupsize.

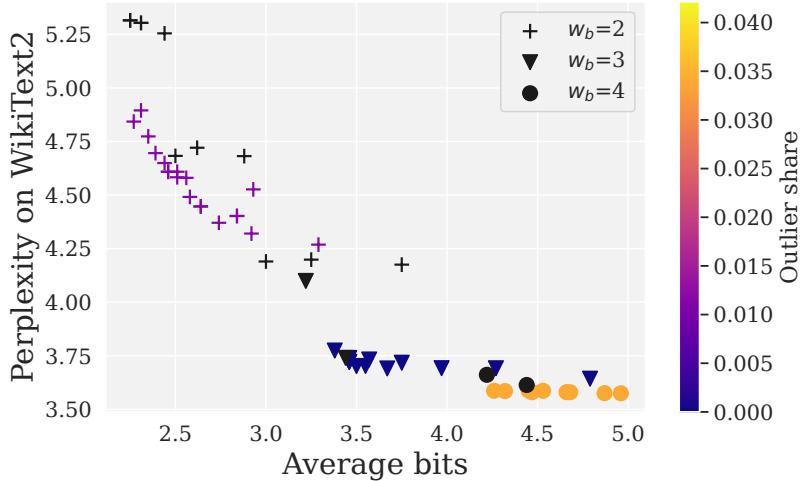


Figure 8: Perplexity of WikiText2 vs average number of bits. Different markers denote different  $b_w$ . Black colors correspond to quantization configurations without outliers and the brightness of the color is proportional to the outlier rate.

OPT					
Size	Method	Avg bits	Wiki2	C4	PTB
6.7B	-	16.00	10.86	11.74	13.09
	SpQR	4.27	10.81	11.88	13.17
	RTN	4	12.10	13.38	16.09
	GPTQ	4	11.39	12.15	13.80
	SpQR	3.94	11.04	11.98	13.33
13B	-	16.00	10.12	11.20	12.34
	SpQR	4.27	10.22	11.27	12.41
	RTN	4	11.32	12.35	15.4
	GPTQ	4	10.31	11.36	12.58
	SpQR	3.93	10.28	11.34	12.52
30B	-	16.00	9.56	10.69	11.84
	SpQR	4.26	9.50	10.73	11.88
	RTN	4	10.97	11.90	14.17
	GPTQ	4	9.63	10.80	11.98
	SpQR	3.94	9.54	10.78	11.93
66B	-	16.00	9.33	10.28	11.36
	SpQR	4.23	9.37	10.32	11.40
	RTN	4	110	249	274
	GPTQ	4	9.55	10.50	11.58
	SpQR	3.91	9.32	10.35	11.43

Table 6: Perplexity on WikiText2 [MXBS16], C4 [RSR<sup>+</sup>20] and Penn Treebank [MKM<sup>+</sup>94] for SpQR and round-to-nearest (RTN) and GPTQ baselines with OPT. We can see that SpQR reaches performances within 1% of the perplexity with less than 4.3 bits per parameter. We also see that for 4-bits per parameter SpQR significantly improves on GPTQ with an improvement as large as the improvement from RTN to GPTQ.

## F Additional results for near-lossless compression

In this section we report the list of quantization configurations for OPT in Table 6 on WikiText2, Penn Treebank, and C4 datasets.

In addition we report results for LM eval harness for LLaMa Table 7. and recently released Falcon models - Falcon-7B and Falcon-40B Table 8.

## G Choice of optimal LLM configuration for specific hardware

In the preceding discussion, we were searching for optimal model configuration given some compression target without targeting any specific hardware or device. However, the question practitioner

**LLaMA**

Size	Method	Avg bits	Winogrande	Piqa	Hellaswag	Arc easy	Arc challenge	Avg score
7B	–	16.00	67.09	78.32	56.41	67.38	38.23	61.492
	SpQR	4.63	67.48	78.45	56.01	67.13	38.23	61.460
	RTN	4	64.72	76.44	53.49	63.51	36.60	58.952
	GPTQ	4	65.35	77.58	54.99	63.55	36.35	59.564
	SpQR	3.45	67.48	78.13	55.27	65.87	38.05	60.960
13B	–	16.00	70.09	78.89	59.11	74.54	43.94	65.314
	SpQR	4.63	69.77	78.94	59.02	74.37	43.17	65.054
	RTN	4	69.61	78.24	57.34	72.56	42.58	64.066
	GPTQ	4	69.06	78.40	58.04	73.23	43.26	64.398
	SpQR	3.45	68.90	78.73	58.22	73.27	42.75	64.374
30B	–	16.00	72.93	80.96	62.66	75.34	46.76	67.730
	SpQR	4.69	72.93	81.01	62.50	76.05	47.18	67.934
	RTN	4	72.06	79.05	60.61	70.66	42.24	64.924
	GPTQ	4	72.61	79.92	61.07	71.8	44.28	65.936
	SpQR	3.49	73.32	80.47	61.96	74.75	46.93	67.486
65B	–	16.00	77.43	81.50	63.95	75.17	47.10	69.030
	SpQR	4.71	76.95	81.56	63.76	75.25	46.93	68.890
	RTN	4	75.14	81.45	62.79	72.64	44.97	67.398
	GPTQ	4	75.85	80.79	62.91	74.20	46.59	68.068
	SpQR	3.52	76.09	81.18	63.54	74.37	45.05	68.046

Table 7: LM eval harness results on LLaMA models.

**Falcon**

Size	Method	Avg bits	Winogrande	Piqa	Hellaswag	Arc easy	Arc challenge	Avg score
7B	–	16.00	67.32	79.49	57.77	74.71	40.10	63.878
	SpQR	4.44	67.09	79.16	57.21	73.86	38.99	63.262
	RTN	4.00	65.51	77.37	51.86	68.69	33.7	59.426
	GPTQ	4.00	66.38	79.11	56.68	73.15	38.48	62.760
	SpQR	3.49	67.88	79.54	57.08	74.03	39.08	63.522
40B	–	16.00	76.62	82.32	64.06	82.03	50.26	71.058
	SpQR	4.46	76.48	82.1	63.8	81.78	50.77	70.986
	RTN	4.00	75.69	80.30	60.52	79.92	49.83	69.252
	GPTQ	4.00	75.93	81.23	63.05	80.85	50.00	70.212
	SpQR	3.45	76.32	81.77	63.70	81.10	49.83	70.544

Table 8: LM eval harness results on Falcon models.

willing to deploy a model for a specific application would ask is: What is the best model and compression setup for a given memory constraint?

In this section, we provide a list of recommendations for the choice of the best LLaMA model and the corresponding compression level that fits into the device memory (RAM or VRAM) without the need of offloading model parameters and activations. We cover a range of available budgets from mobile devices to high-end workstation GPUs. Recommendations are presented in Table 9.

Device	Memory (GiB)	LLaMA	$\bar{b}$
iPhone13	4	7B	$\leq 3.5$
iPhone14	6	7B 13B	$\simeq 4.5$ $\leq 3.5$
Consumer laptop	8	13B	$\leq 4$
RTX4070	10-12	14B	$\simeq 4.5$
RTX4080	16	30B	$\leq 4$
RTX4090	24	30B	$\simeq 4.5$
V100	32	65B	$\leq 3.5$
A6000	48	65B	$\simeq 4.5$

Table 9: Choice of the best LLaMA for a given memory constraint.

## H Sensitivity to random seed

The experiments we report throughout Section 5 use one fixed random seed (the default value from the supplementary code). To verify that our results are robust to randomness, we run SpQR with 5 random seeds (0-5) and measure the adjusted standard deviation.

For this evaluation, we compress LLaMA-65B with SpQR using  $b_w = b_z = b_s = 3$  and  $\beta_1 = \beta_2 = 16$ , which corresponds to 3.625 bits per parameter. The resulting perplexity scores are  $3.75 \pm 0.003$  (WikiText2),  $7.03 \pm 0.01$  (Penn Treebank) and  $5.75 \pm 0.00086$  (C4). In addition to the chosen random seed, these standard deviations can be affected by the inherent nondeterminism of GPU computation. Overall, the standard deviations are at least one order of magnitude smaller than the difference between SpQR, GPTQ, and RTN.

## I Generative examples

Finally, we showcase several examples of how SpQR quantization affects the generated samples. For this evaluation, we take several prompts and use the compressed language model to continue generating text from these prompts. We compare the original LLaMA-65B and two quantized versions: SpQR and RTN-4bit. More specifically, we use the SpQR configuration that corresponds to near-lossless compression from Table 1. We use greedy autoregressive inference for all generated samples to ensure reproducibility. The examples in Figure 9 show that all models produce a valid text, but SpQR matches the 16-bit model more frequently. The near-lossless algorithm also seems to produce more semantically similar texts.

## J Broader impact

Our method enables the deployment of high-quality LLMs in the 7-13B parameters range to memory-limited devices such as laptops and phones. With our method, it is possible to develop specialized 7B LLMs in hassle-free 16-bit and then enable the deployment of such LLMs to phones by applying SpQR. Since SpQR is practically lossless, this ensures a reliable performance level for deployed LLMs which is important for consumer applications. Since mobile phones are ubiquitous and LLMs

Prompt:	Every time data scientist Kim ran her new algorithm, it kept outputting the same unexpected result: a recipe for
16bit:	iced tea. Kim was working on a project to predict the likelihood of a patient being readmitted to the hospital within 30 days of discharge. She was using a machine learning algorithm to analyze the data, and the algorithm
SpQR:	iced tea. Kim was working on a project to predict the likelihood of a patient being readmitted to the hospital within 30 days of discharge. She had been given a dataset of 100,000
RTN 4-bit:	iced tea. "I'm not sure what's going on," she said. "I've been running it for days, and it keeps giving me the same result." "I'm not sure what's going on
Prompt:	Data scientist Grace was working late one night when she discovered a glitch in the matrix of her data. It spelled out
16bit:	100% proof that the world was going to end in 10 days. She tried to tell her boss, but he didn't believe her. She tried to tell her friends, but they didn't believe her
SpQR:	100% proof that the world was going to end in 10 days. She tried to tell her boss, but he just laughed and said, "I'm not going to believe in some data glitch." She
RTN 4-bit:	100% in binary code. She was so excited she ran to tell her boss, but he was busy. She ran to tell her colleague, but he was busy. She ran to tell her friend, but she
Prompt:	Last Tuesday, just as the sun was setting, my neighbor's dog, Rover, started to speak to me in perfect French. He said
16bit:	"Je suis un chien." I was shocked. I said, "Moi aussi." I'm not sure what to make of this. I'm not sure what to make of anything. I'm not sure what
SpQR:	"Je suis un chien." I was shocked. I said, "Rover, you can speak French?" He said, "Oui, je parle français." I said, "How did you learn to speak French?"
RTN 4-bit:	"Bonjour, Monsieur. Je m'appelle Rover. Je suis un chien. Je suis un chien français. Je suis un chien français qui parle français. Je suis un chien français qui parle français
Prompt:	In a parallel universe where cats are the rulers of Earth, our feline king was pondering his first decree
16bit:	" to make all dogs illegal. "I'm not sure that's a good idea," said his chief adviser, a cat named Whiskers. "Why not?" asked the king. "Because,
SpQR:	" to make all dogs illegal. "I'm not sure that's a good idea," said his chief adviser, a cat named Whiskers. "Why not?" asked the king. "Because,
RTN 4-bit:	" to make the world a better place for cats. He was about to sign the decree when he was interrupted by a knock on the door. "Come in," he said. The door opened and a cat entered.

Figure 9: Texts generated by different quantized LLaMA-65B models with the same prompt.

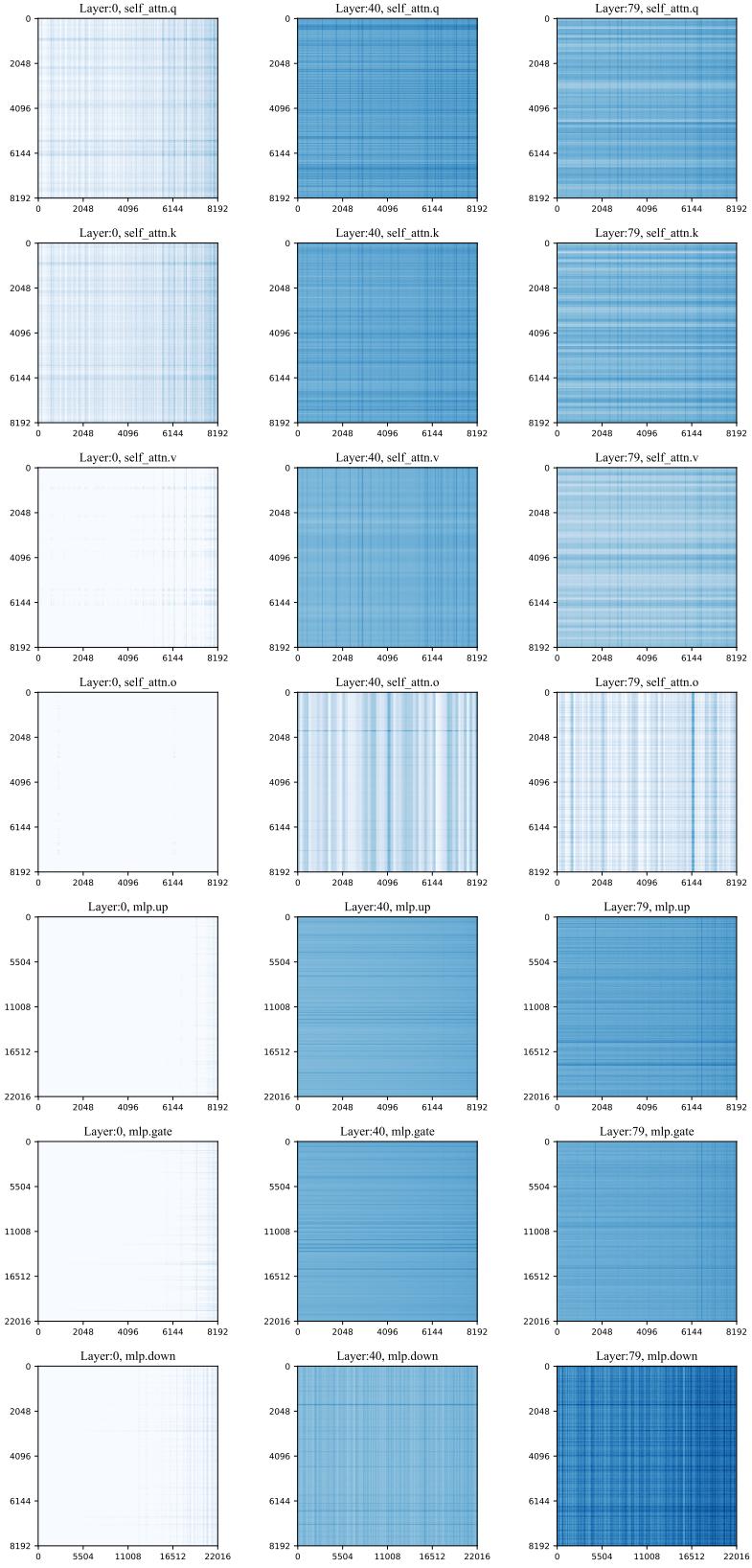


Figure 10: A grid of weight log-sensitivities for LLaMA-65B for 3-bit GPTQ compression with per-row quantization statistics. Each row corresponds to a specific layer type (e.g. attention query, mlp gate), and the columns represent layer depth.

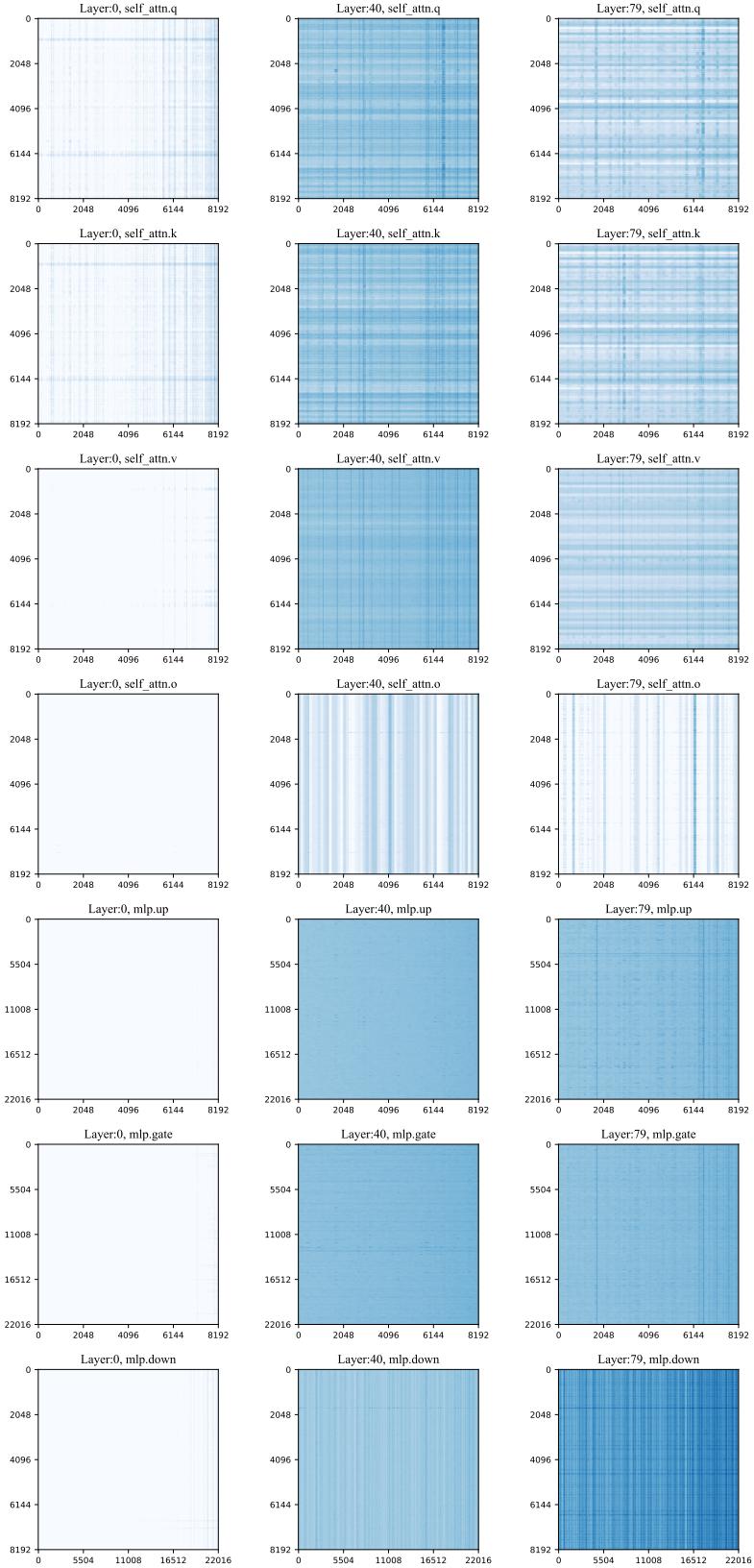


Figure 11: A grid of weight log-sensitivities for LLaMA-65B for 3-bit GPTQ compression with group-wise quantization of block size 128. Each row corresponds to a specific layer type (e.g. attention query, mlp gate), and the columns represent layer depth.

powerful general-purpose tools, SpQR might have a wide-reaching effect on how LLMs are used by the general population to complete useful tasks.

LLMs are inherently a dual-use technology that can bring both significant benefits and serious harm. The ethical and societal risks of LLMs range from deliberate malicious use (e.g. generating spam) and accidental misuse to adverse economic side-effects [WMR<sup>+</sup>21]. However, we believe that the marginal impact of SpQR will be positive or neutral since the LLMs we use are already openly available. Better quantization algorithms like SpQR let users with low-end devices run larger and generally more accurate language models. In other words, our algorithm does not create models with new capabilities (and risks): it only makes existing models more accessible.

## K On the use of LLMs in this work

Following the request in this year’s call for papers, we describe the use of large language models in our paper. We used two different chat-based language models: ChatGPT and Claude+. We used these models to accelerate the process of writing LaTeX code in Alg. 1 and Figure 3 (via Tikz). We also used these LLMs to provide slight improvements to the table design throughout the paper.

In addition to this, we use ChatGPT to generate some prompts for Appendix I. Finally, we used Claude+ to produce possible formulations for the outlier criterion in Alg. 1. In all these cases, we used LLMs through chat-based user interfaces, instructing them to generate code (LaTeX) or suggest improvements. If the suggested changes would not work as expected, we reported them to the model in natural language, using the same chat-based interface.

# EFFICIENT STREAMING LANGUAGE MODELS WITH ATTENTION SINKS

Guangxuan Xiao<sup>1\*</sup> Yuandong Tian<sup>2</sup> Beidi Chen<sup>3</sup> Song Han<sup>1</sup> Mike Lewis<sup>2</sup>

<sup>1</sup> Massachusetts Institute of Technology

<sup>2</sup> Meta AI

<sup>3</sup> Carnegie Mellon University

<https://github.com/mit-han-lab/streaming-llm>

## ABSTRACT

Deploying Large Language Models (LLMs) in streaming applications such as multi-round dialogue, where long interactions are expected, is urgently needed but poses two major challenges. Firstly, during the decoding stage, caching previous tokens’ Key and Value states (KV) consumes extensive memory. Secondly, popular LLMs cannot generalize to longer texts than the training sequence length. Window attention, where only the most recent KVs are cached, is a natural approach — but we show that it fails when the text length surpasses the cache size. We observe an interesting phenomenon, namely *attention sink*, that keeping the KV of initial tokens will largely recover the performance of window attention. In this paper, we first demonstrate that the emergence of *attention sink* is due to the strong attention scores towards initial tokens as a “sink” even if they are not semantically important. Based on the above analysis, we introduce StreamingLLM, an efficient framework that enables LLMs trained with a *finite length* attention window to generalize to *infinite sequence length* without any fine-tuning. We show that StreamingLLM can enable Llama-2, MPT, Falcon, and Pythia to perform stable and efficient language modeling with up to 4 million tokens and more. In addition, we discover that adding a placeholder token as a dedicated attention sink during pre-training can further improve streaming deployment. In streaming settings, StreamingLLM outperforms the sliding window recomputation baseline by up to  $22.2 \times$  speedup. Code and datasets are provided in the link.

## 1 INTRODUCTION

Large Language Models (LLMs) (Radford et al., 2018; Brown et al., 2020; Zhang et al., 2022; OpenAI, 2023; Touvron et al., 2023a;b) are becoming ubiquitous, powering many natural language processing applications such as dialog systems (Schulman et al., 2022; Taori et al., 2023; Chiang et al., 2023), document summarization (Goyal & Durrett, 2020; Zhang et al., 2023a), code completion (Chen et al., 2021; Rozière et al., 2023) and question answering (Kamalloo et al., 2023). To unleash the full potential of pretrained LLMs, they should be able to efficiently and accurately perform long sequence generation. For example, an ideal ChatBot assistant can stably work over the content of recent day-long conversations. However, it is very challenging for LLM to generalize to longer sequence lengths than they have been pretrained on, e.g., 4K for Llama-2 Touvron et al. (2023b).

The reason is that LLMs are constrained by the attention window during pre-training. Despite substantial efforts to expand this window size (Chen et al., 2023; kaiokendev, 2023; Peng et al., 2023) and improve training (Dao et al., 2022; Dao, 2023) and inference (Pope et al., 2022; Xiao et al., 2023; Anagnostidis et al., 2023; Zhang et al., 2023b) efficiency for lengthy inputs, the acceptable sequence length remains intrinsically *finite*, which doesn’t allow persistent deployments.

In this paper, we first introduce the concept of LLM streaming applications and ask the question:

*Can we deploy an LLM for infinite-length inputs without sacrificing efficiency and performance?*

---

\*Part of the work done during an internship at Meta AI.

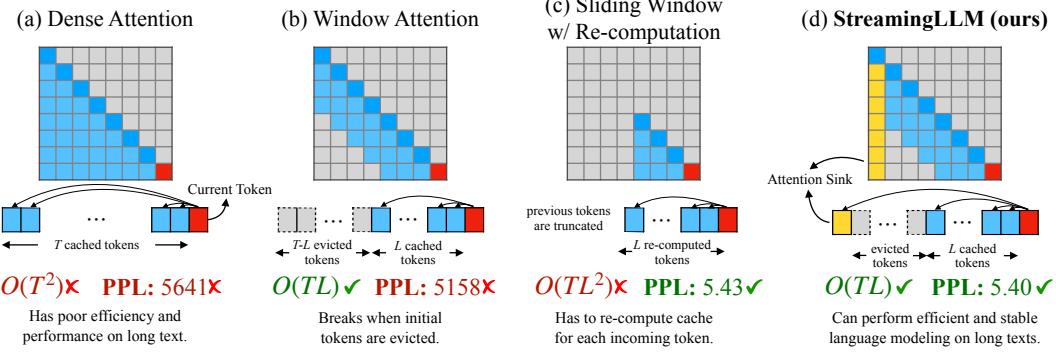


Figure 1: **Illustration of StreamingLLM vs. existing methods.** The language model, pre-trained on texts of length  $L$ , predicts the  $T$ th token ( $T \gg L$ ). (a) Dense Attention has  $O(T^2)$  time complexity and an increasing cache size. Its performance decreases when the text length exceeds the pre-training text length. (b) Window Attention caches the most recent  $L$  tokens’ KV. While efficient in inference, performance declines sharply once the starting tokens’ keys and values are evicted. (c) Sliding Window with Re-computation rebuilds the KV states from the  $L$  recent tokens for each new token. While it performs well on long texts, its  $O(TL^2)$  complexity, stemming from quadratic attention in context re-computation, makes it considerably slow. (d) StreamingLLM keeps the *attention sink* (several initial tokens) for stable attention computation, combined with the recent tokens. It’s efficient and offers stable performance on extended texts. Perplexities are measured using the Llama-2-13B model on the first book (65K tokens) in the PG-19 test set.

When applying LLMs for infinite input streams, two primary challenges arise:

1. During the decoding stage, Transformer-based LLMs cache the Key and Value states (KV) of all previous tokens, as illustrated in Figure 1 (a), which can lead to excessive memory usage and increasing decoding latency (Pope et al., 2022).
2. Existing models have limited length extrapolation abilities, i.e., their performance degrades (Press et al., 2022; Chen et al., 2023) when the sequence length goes beyond the attention window size set during pre-training.

An intuitive approach, known as window attention (Beltagy et al., 2020) (Figure 1 b), maintains only a fixed-size sliding window on the KV states of most recent tokens. Although it ensures constant memory usage and decoding speed after the cache is initially filled, the model collapses once the sequence length exceeds the cache size, i.e., *even just evicting the KV of the first token*, as illustrated in Figure 3. Another strategy is the sliding window with re-computation (shown in Figure 1 c), which rebuilds the KV states of recent tokens for each generated token. While it offers strong performance, this approach is significantly slower due to the computation of quadratic attention within its window, making this method impractical for real-world streaming applications.

To understand the failure of window attention, we find an interesting phenomenon of autoregressive LLMs: a surprisingly large amount of attention score is allocated to the initial tokens, irrespective of their relevance to the language modeling task, as visualized in Figure 2. We term these tokens “**attention sinks**”. Despite their lack of semantic significance, they collect significant attention scores. We attribute the reason to the Softmax operation, which requires attention scores to sum up to one for all contextual tokens. Thus, even when the current query does not have a strong match in many previous tokens, the model still needs to allocate these unneeded attention values somewhere so it sums up to one. The reason behind *initial* tokens as sink tokens is intuitive: initial tokens are visible to almost all subsequent tokens because of the autoregressive language modeling nature, making them more readily trained to serve as attention sinks.

Based on the above insights, we propose StreamingLLM, a simple and efficient framework that enables LLMs trained with a finite attention window to work on text of infinite length without fine-tuning. StreamingLLM exploits the fact that attention sinks have high attention values, and preserving them can maintain the attention score distribution close to normal. Therefore, StreamingLLM simply keeps the attention sink tokens’ KV (with just 4 initial tokens sufficing) together with the sliding window’s KV to anchor the attention computation and stabilize the model’s performance. With StreamingLLM, models including Llama-2-[7, 13, 70]B, MPT-[7, 30]B, Falcon-[7, 40]B, and Pythia-

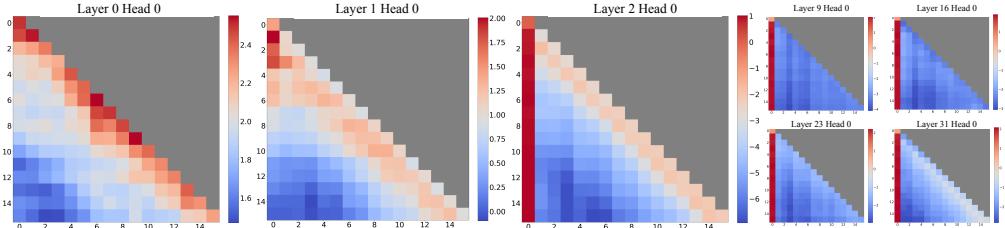


Figure 2: Visualization of the *average* attention logits in Llama-2-7B over 256 sentences, each with a length of 16. Observations include: (1) The attention maps in the first two layers (layers 0 and 1) exhibit the "local" pattern, with recent tokens receiving more attention. (2) Beyond the bottom two layers, the model heavily attends to the initial token across all layers and heads.

[2.9,6.9,12]B can reliably model 4 million tokens, and potentially even more. Compared with the only viable baseline, sliding window with recomputation, StreamingLLM achieves up to  $22.2 \times$  speedup, realizing the streaming use of LLMs.

Finally, we confirm our attention sink hypothesis and demonstrate that language models can be pre-trained to require only a single attention sink token for streaming deployment. Specifically, we suggest that an extra learnable token at the beginning of all training samples can serve as a designated attention sink. By pre-training 160-million parameter language models from scratch, we demonstrate that adding this single sink token preserves the model's performance in streaming cases. This stands in contrast to vanilla models, which necessitate the reintroduction of multiple initial tokens as attention sinks to achieve the same performance level.

## 2 RELATED WORK

Extensive research has been done on applying LLMs to lengthy texts, with three main areas of focus: **Length Extrapolation**, **Context Window Extension**, and **Improving LLMs' Utilization of Long Text**. While seemingly related, it's worth noting that progress in one direction doesn't necessarily lead to progress in the other. For example, extending the context size of LLMs doesn't improve the model's performance beyond the context size, and neither approach ensures effective use of the long context. Our StreamingLLM framework primarily lies in the first category, where LLMs are applied to text significantly exceeding the pre-training window size, potentially even of infinite length. We do not expand the attention window size of LLMs or enhance the model's memory and usage on long texts. The last two categories are orthogonal to our focus and could be integrated with our techniques.

**Length extrapolation** aims to enable language models trained on shorter texts to handle longer ones during testing. A predominant avenue of research targets the development of relative position encoding methods for Transformer models, enabling them to function beyond their training window. One such initiative is Rotary Position Embeddings (RoPE) (Su et al., 2021), which transforms the queries and keys in every attention layer for relative position integration. Despite its promise, subsequent research (Press et al., 2022; Chen et al., 2023) indicated its underperformance on text that exceeds the training window. Another approach, ALiBi (Press et al., 2022), biases the query-key attention scores based on their distance, thereby introducing relative positional information. While this exhibited improved extrapolation, our tests on MPT models highlighted a breakdown when the text length was vastly greater than the training length. Current methodologies, however, have yet to achieve infinite length extrapolation, causing no existing LLMs to fit for streaming applications.

**Context Window Extension** centers on expanding the LLMs' context window, enabling the processing of more tokens in one forward pass. A primary line of work addresses the training efficiency problem. Given the attention to computation's quadratic complexity during training, developing a long-context LLM is both a computational and memory challenge. Solutions have ranged from system-focused optimizations like FlashAttention (Dao et al., 2022; Dao, 2023), which accelerates attention computation and reduces memory footprint, to approximate attention methods (Zaheer et al., 2020; Beltagy et al., 2020; Wang et al., 2020; Kitaev et al., 2020) that trade model quality for efficiency. Recently, there has been a surge of work on extending pre-trained LLMs with RoPE (Chen et al., 2023; kaiokendev, 2023; bloc97, 2023; Peng et al., 2023), involving position interpolation and fine-tuning. However, all the aforementioned techniques only extend LLMs' context window to a limited extent, which falls short of our paper's primary concern of handling limitless inputs.

**Improving LLMs' Utilization of Long Text** optimizes LLMs to better capture and employ the content within the context rather than merely taking them as inputs. As highlighted by Liu et al.

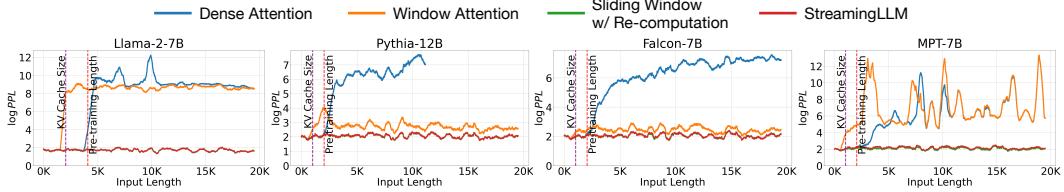


Figure 3: Language modeling perplexity on texts with 20K tokens across various LLM. Observations reveal consistent trends: (1) Dense attention fails once the input length surpasses the pre-training attention window size. (2) Window attention collapses once the input length exceeds the cache size, i.e., the initial tokens are evicted. (3) StreamingLLM demonstrates stable performance, with its perplexity nearly matching that of the sliding window with re-computation baseline.

and Li et al., success in the previously mentioned two directions does not necessarily translate to competent utilization of lengthy contexts. Addressing this effective usage of prolonged contexts within LLMs is still a challenge. Our work concentrates on stably harnessing the most recent tokens, enabling the seamless streaming application of LLMs.

### 3 STREAMINGLLM

#### 3.1 THE FAILURE OF WINDOW ATTENTION AND ATTENTION SINKS

While the window attention technique offers efficiency during inference, it results in an exceedingly high language modeling perplexity. Consequently, the model’s performance is unsuitable for deployment in streaming applications. In this section, we use the concept of *attention sink* to explain the failure of window attention, serving as the inspiration behind StreamingLLM.

**Identifying the Point of Perplexity Surge.** Figure 3 shows the perplexity of language modeling on a 20K token text. It is evident that perplexity spikes when the text length surpasses the cache size, led by the exclusion of initial tokens. This suggests that the initial tokens, regardless of their distance from the tokens being predicted, are crucial for maintaining the stability of LLMs.

**Why do LLMs break when removing *initial tokens’ KV*?** We visualize attention maps from all layers and heads of the Llama-2-7B and models in Figure 2. We find that, beyond the bottom two layers, the model consistently focuses on the initial tokens across all layers and heads. The implication is clear: removing these initial tokens’ KV will remove a considerable portion of the denominator in the SoftMax function (Equation 1) in attention computation. This alteration leads to a significant shift in the distribution of attention scores away from what would be expected in normal inference settings.

$$\text{SoftMax}(x)_i = \frac{e^{x_i}}{e^{x_1} + \sum_{j=2}^N e^{x_j}}, \quad x_1 \gg x_j, j \in 2, \dots, N \quad (1)$$

There are two possible explanations for the importance of the initial tokens in language modeling: (1) Either their semantics are crucial, or (2) the model learns a bias towards their absolute position. To distinguish between these possibilities, we conduct experiments (Table 1), wherein the first four tokens are substituted with the linebreak token “\n”. The observations indicate that the model still significantly emphasizes these initial linebreak tokens. Furthermore, reintroducing them restores the language modeling perplexity to levels comparable to having the original initial tokens. This suggests that the absolute position of the starting tokens, rather than their semantic value, holds greater significance.

**LLMs attend to Initial Tokens as Attention Sinks.** To explain why the model disproportionately focuses on initial tokens—regardless of their semantic relevance to language modeling, we introduce the concept of “*attention sink*”. The nature of the SoftMax function (Equation 1) prevents all attended tokens from having zero values. This requires aggregating some information from other tokens across all heads in all layers, even if the current embedding has sufficient self-contained information for its prediction. Consequently, the model tends to dump unnecessary attention values to specific tokens. A similar observation has been made in the realm of quantization outliers (Xiao et al., 2023; Bondarenko et al., 2023), leading to the proposal of SoftMax-Off-by-One (Miller, 2023) as a potential remedy.

Table 1: Window attention has poor performance on long text. The perplexity is restored when we reintroduce the initial four tokens alongside the recent 1020 tokens (4+1020). Substituting the original four initial tokens with linebreak tokens “\n” (4\n+1020) achieves comparable perplexity restoration. Cache config x+y denotes adding x initial tokens with y recent tokens. Perplexities are measured on the first book (65K tokens) in the PG19 test set.

Llama-2-13B	PPL (↓)
0 + 1024 (Window)	5158.07
4 + 1020	5.40
4\n+1020	5.60

Table 2: Effects of reintroduced initial token numbers on StreamingLLM. (1) Window attention (0+y) has a drastic increase in perplexity. (2) Introducing one or two initial tokens usually doesn’t suffice to fully restore model perplexity, indicating that the model doesn’t solely use the first token as the attention sink. (3) Introducing four initial tokens generally suffices; further additions have diminishing returns. Cache config x+y denotes adding x initial tokens to y recent tokens. Perplexities are evaluated on 400K tokens in the concatenated PG19 test set.

Cache Config	0+2048	1+2047	2+2046	4+2044	8+2040
Falcon-7B	17.90	12.12	12.12	12.12	12.12
MPT-7B	460.29	14.99	15.00	14.99	14.98
Pythia-12B	21.62	11.95	12.09	12.09	12.02
Cache Config	0+4096	1+4095	2+4094	4+4092	8+4088
Llama-2-7B	3359.95	11.88	10.51	9.59	9.54

Why do various autoregressive LLMs, such as Llama-2, MPT, Falcon, and Pythia, consistently focus on *initial tokens* as their attention sinks, rather than other tokens? Our explanation is straightforward: Due to the sequential nature of autoregressive language modeling, initial tokens are visible to all subsequent tokens, while later tokens are only visible to a limited set of subsequent tokens. As a result, initial tokens are more easily trained to serve as attention sinks, capturing unnecessary attention.

We’ve noted that LLMs are typically trained to utilize multiple initial tokens as attention sinks rather than just one. As illustrated in Figure 2, the introduction of four initial tokens, as attention sinks, suffices to restore the LLM’s performance. In contrast, adding just one or two doesn’t achieve full recovery. We believe this pattern emerges because these models didn’t include a consistent starting token across all input samples during pre-training. Although Llama-2 does prefix each paragraph with a “<s>” token, it’s applied before text chunking, resulting in a mostly random token occupying the zeroth position. This lack of a uniform starting token leads the model to use several initial tokens as attention sinks. We hypothesize that by incorporating a stable learnable token at the start of all training samples, it could singularly act as a committed attention sink, eliminating the need for multiple initial tokens to ensure consistent streaming. We will validate this hypothesis in Section 3.3.

### 3.2 ROLLING KV CACHE WITH ATTENTION SINKS

To enable LLM streaming in already trained LLMs, we propose a straightforward method that can recover window attention’s perplexity without any model finetuning. Alongside the current sliding window tokens, we reintroduce a few starting tokens’ KV in the attention computation. The KV cache in StreamingLLM can be conceptually divided into two parts, as illustrated in Figure 4: (1) Attention sinks (four initial tokens) stabilize the attention computation; (2) Rolling KV Cache retains the most recent tokens, crucial for language modeling. StreamingLLM’s design is versatile and can be seamlessly incorporated into any autoregressive language model that employs relative positional encoding, such as RoPE (Su et al., 2021) and ALiBi (Press et al., 2022).

When determining the relative distance and adding positional information to tokens, StreamingLLM focuses on positions *within the cache* rather than those *in the original text*. This distinction is crucial for StreamingLLM’s performance. For instance, if the current cache has tokens [0, 1, 2, 3, 6, 7, 8] and is in the process of decoding the 9th token, the positions assigned are [0, 1, 2, 3, 4, 5, 6, 7], rather than the positions in the original text, which would be [0, 1, 2, 3, 6, 7, 8, 9].

For encoding like RoPE, we cache the Keys of tokens *prior to* introducing the rotary transformation. Then, we apply position transformation to the keys in the rolling cache at each decoding phase. On the other hand, integrating with ALiBi is more direct. Here, the contiguous linear bias is applied instead of a ‘jumping’ bias to the attention scores. This method of assigning positional embedding within the cache is crucial to StreamingLLM’s functionality, ensuring that the model operates efficiently even beyond its pre-training attention window size.

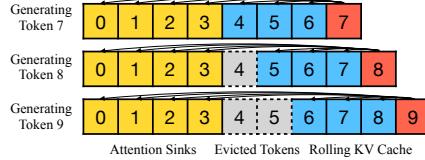


Figure 4: The KV cache of StreamingLLM. To enable LLM streaming in already trained LLMs, we propose a straightforward method that can recover window attention’s perplexity without any model finetuning. Alongside the current sliding window tokens, we reintroduce a few starting tokens’ KV in the attention computation. The KV cache in StreamingLLM can be conceptually divided into two parts, as illustrated in Figure 4: (1) Attention sinks (four initial tokens) stabilize the attention computation; (2) Rolling KV Cache retains the most recent tokens, crucial for language modeling. StreamingLLM’s design is versatile and can be seamlessly incorporated into any autoregressive language model that employs relative positional encoding, such as RoPE (Su et al., 2021) and ALiBi (Press et al., 2022).

### 3.3 PRE-TRAINING LLMs WITH ATTENTION SINKS

As elaborated in Section 3.1, a significant reason for the model’s excessive attention to multiple initial tokens is the absence of a designated sink token to offload excessive attention scores. Due to this, the model inadvertently designates globally visible tokens, primarily the initial ones, as attention sinks. A potential remedy can be the intentional inclusion of a global trainable attention sink token, denoted as a “Sink Token”, which would serve as a repository for unnecessary attention scores. Alternatively, replacing the conventional SoftMax function with a variant like SoftMax-off-by-One (Miller, 2023),

$$\text{SoftMax}_1(x)_i = \frac{e^{x_i}}{1 + \sum_{j=1}^N e^{x_j}}, \quad (2)$$

which does not require the attention scores on all contextual tokens to sum up to one, might also be effective. Note that this SoftMax alternative is equivalent to using a token with an all-zero Key and Value features in the attention computation. We denote this method as “Zero Sink” to fit it consistently in our framework.

For validation, we pre-train three language models with 160 million parameters from scratch under identical settings. The first model utilizes the standard SoftMax attention (Vanilla), the second replaced the regular attention mechanism with SoftMax<sub>1</sub> (Zero Sink), and one prepending a learnable placeholder token (Sink Token) in all training samples. As shown in Table 3, while the zero sink alleviates the attention sink problem to some extent, the model still relies on other initial tokens as attention sinks. Introducing a sink token is highly effective in stabilizing the attention mechanism. Simply pairing this sink token with recent tokens sufficiently anchors the model’s performance, and the resulting evaluation perplexity is even marginally improved. Given these findings, we recommend training future LLMs with a sink token in all samples to optimize streaming deployment.

## 4 EXPERIMENTS

We evaluate StreamingLLM using four prominent recent model families: Llama-2 (Touvron et al., 2023b), MPT (Team, 2023), PyThia (Biderman et al., 2023), and Falcon (Almazrouei et al., 2023). Notably, Llama-2, Falcon, and Pythia incorporate RoPE (Su et al., 2021), whereas MPT employs ALiBi (Press et al., 2022) — two of the most influential position encoding techniques in recent research. Our diverse model selection ensures the validity and robustness of our findings. We benchmark StreamingLLM against established baselines such as dense attention, window attention, and the sliding window approach with re-computation. In all subsequent experiments with StreamingLLM, we default to using four initial tokens as attention sinks unless stated otherwise.

### 4.1 LANGUAGE MODELING ON LONG TEXTS ACROSS LLM FAMILIES AND SCALES

We firstly evaluate StreamingLLM’s language modeling perplexity using the concatenated PG19 (Rae et al., 2020) test set, which contains 100 long books. For Llama-2 models, the cache size is set at 2048, while for Falcon, Pythia, and MPT models, it’s set at 1024. This is half the pre-training window size chosen to enhance visualization clarity.

Figure 3 illustrates that StreamingLLM can match the oracle baseline (sliding window with re-computation) in terms of perplexity on texts spanning 20K tokens. Meanwhile, the dense attention technique fails when the input length exceeds its pre-training window, and the window attention technique struggles when the input length surpasses the cache size, leading to the eviction of the initial tokens. In Figure 5, we further substantiate that StreamingLLM can reliably handle exceptionally extended texts, encompassing more than 4 million tokens, across a spectrum of model families and scales. This includes Llama-2-[7,13,70]B, Falcon-[7,40]B, Pythia-[2.8,6.9,12]B, and MPT-[7,30]B.

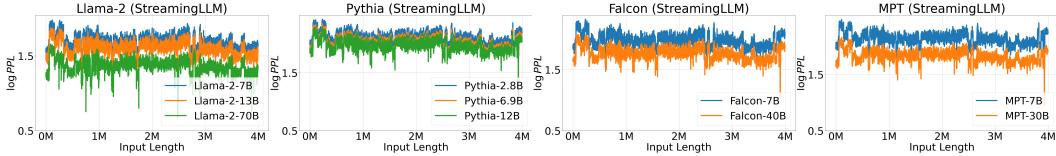


Figure 5: Language modeling perplexity of StreamingLLM for super long texts with 4 million tokens across various LLM families and model scales. The perplexity remains stable throughout. We use the concatenated test set of PG19 (100 books) to perform language modeling, with perplexity fluctuations attributable to the transition between books.

#### 4.2 RESULTS OF PRE-TRAINING WITH A SINK TOKEN

To validate our suggestion that introducing a sink token to all pre-training samples improves streaming LLMs, we trained two language models, each with 160 million parameters, under identical conditions. While one model adhered to the original training settings, the other incorporated a sink token at the start of every training sample. Our experiments employed the Pythia-160M (Biderman et al., 2023) codebase and followed its training recipe. We train the models on an 8xA6000 NVIDIA GPU server using the deduplicated Pile (Gao et al., 2020) dataset. Apart from reducing the training batch size to 256, we retained all Pythia training configurations, including learning rate schedules, model initialization, and dataset permutations. Both models were trained for 143,000 steps.

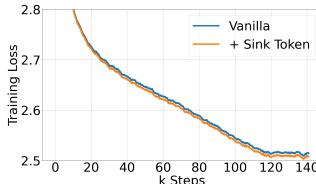


Figure 6: Pre-training loss curves of models w/ and w/o sink tokens. Two models have a similar convergence trend.

Table 4: Zero-shot accuracy (in %) across 7 NLP benchmarks, including ARC-[Challenge, Easy], HellaSwag, LAMBADA, OpenbookQA, PIQA, and Winogrande. The inclusion of a sink token during pre-training doesn't harm the model performance.

Methods	ARC-c	ARC-e	HS	LBD	OBQA	PIQA	WG
Vanilla	18.6	45.2	29.4	39.6	16.0	62.2	50.1
+Sink Token	<b>19.6</b>	<b>45.6</b>	<b>29.8</b>	<b>39.9</b>	<b>16.6</b>	<b>62.6</b>	<b>50.8</b>

**Convergence and Normal Model Performance.** Including a sink token during pre-training has no negative impact on model convergence and subsequent performance on a range of NLP benchmarks. As depicted in Figure 6, models trained with a sink token exhibit similar convergence dynamics compared to their vanilla counterparts. We evaluate the two models on seven diverse NLP benchmarks, including ARC-[Challenge, Easy] (Clark et al., 2018), HellaSwag (Zellers et al., 2019), LAMBADA (Paperno et al., 2016), OpenbookQA (Mihaylov et al., 2018), PIQA (Bisk et al., 2020), and Winogrande (Sakaguchi et al., 2019). As shown in Table 4, the model pre-trained with a sink token performs similarly to that trained using the vanilla approach.

**Streaming Performance.** As illustrated in Table 3, the streaming perplexities differ between models trained using traditional methods and those augmented with a sink token. Remarkably, the vanilla model requires the addition of multiple tokens as attention sinks to maintain stable streaming perplexity. In contrast, the model trained with a sink token achieves satisfactory streaming performance using just the sink token.

**Attention Visualization.** Figure 7 contrasts attention maps for models pre-trained with and without a sink token. The model without the sink token, similar to Llama-2-7B (Figure 2), shows early-layer local attention and deeper-layer focus on initial tokens. In contrast, models trained with a sink token consistently concentrate on the sink across layers and heads, indicating an effective attention offloading mechanism. This strong focus on the sink, with reduced attention to other initial tokens, explains the sink token's efficacy in enhancing model's streaming performance.

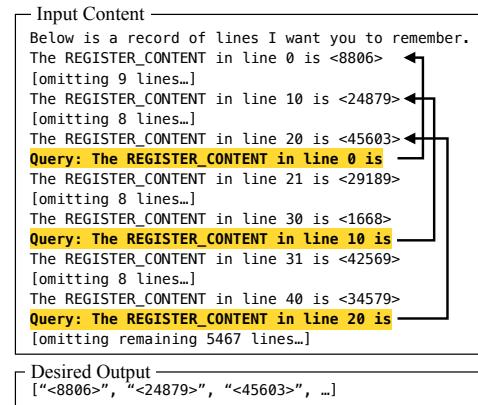


Figure 8: The first sample in StreamEval.

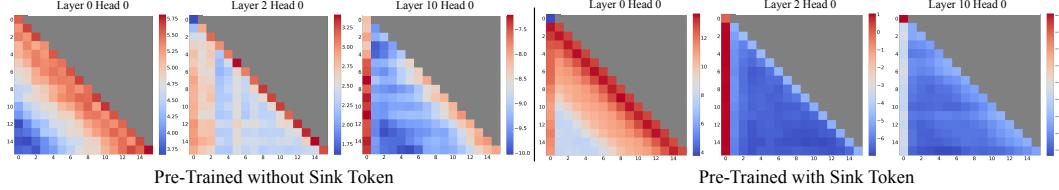


Figure 7: Visualization of average attention logits over 256 sentences, each 16 tokens long, comparing models pre-trained with (left) and without (right) a sink token. Both maps show the same layers and heads. Key observations: (1) Without a sink token, models show local attention in lower layers and increased attention to initial tokens in deeper layers. (2) With a sink token, there is clear attention directed at it across all layers, effectively collecting redundant attention. (3) With the presence of the sink token, less attention is given to other initial tokens, supporting the benefit of designating the sink token to enhance the streaming performance.

Table 5: Accuracy (in %) on the ARC-[Easy, Challenge] datasets. Questions were concatenated and answered in a streaming manner to mimic a real-world chat setting. The dense baseline fails due to Out-of-Memory (OOM) errors. Window attention has poor accuracy. StreamingLLM has comparable results with the one-shot sample-by-sample baseline. Window attention and StreamingLLM use cache sizes of 1024.

Model	Llama-2-7B-Chat		Llama-2-13B-Chat		Llama-2-70B-Chat	
Dataset	Arc-E	Arc-C	Arc-E	Arc-C	Arc-E	Arc-C
One-shot	71.25	53.16	78.16	63.31	91.29	78.50
Dense			OOM			
Window	3.58	1.39	0.25	0.34	0.12	0.32
StreamingLLM	71.34	55.03	80.89	65.61	91.37	80.20

#### 4.3 RESULTS ON STREAMING QUESTION ANSWERING WITH INSTRUCTION-TUNED MODELS

To show StreamingLLM’s real-world applicability, we emulate multi-round question-answering using instruction-tuned LLMs, commonly used in real-world scenarios.

We first concatenate all question-answer pairs from the ARC-[Challenge, Easy] datasets, feed the continuous stream to Llama-2-[7,13,70]B-Chat models, and assess model completions at each answer position using an exact match criterion. As table 5 indicates, dense attention results in Out-of-Memory (OOM) errors, showing it unsuitable for this setting. While the window attention method works efficiently, it exhibits low accuracy due to random outputs when the input length exceeds the cache size. Conversely, StreamingLLM excels by efficiently handling the streaming format, aligning with the one-shot, sample-by-sample baseline accuracy.

Highlighting a more fitting scenario for StreamingLLM, we introduce a dataset, StreamEval, inspired by the LongEval (Li et al., 2023) benchmark. As depicted in Figure 8, diverging from LongEval’s single query over a long-span setup, we query the model every 10 lines of new information. Each query’s answer is consistently 20 lines prior, reflecting real-world instances where questions typically pertain to recent information. As illustrated in Figure 9, LLMs employing StreamingLLM maintain reasonable accuracy even as input lengths approach 120K tokens. In contrast, both dense and window attention fail at the pre-training text length and the KV cache size, respectively. Additionally, we utilize two context-extended models, LongChat-7b-v1.5-32k (Li et al., 2023) and Llama-2-7B-32K-Instruct (Together, 2023), to show that StreamingLLM can complement context extension techniques. Within StreamingLLM, context extension means broadening the maximum cache size of streaming LLMs, enabling the capture of broader local information.

#### 4.4 ABLATION STUDIES

**Numbers of Initial Tokens.** In Table 2, we ablate the effect of adding varying numbers of initial tokens with recent tokens on the streaming perplexity. The results show the insufficiency of introducing merely one or two initial tokens, whereas a threshold of four initial tokens appears enough, with subsequent additions contributing marginal effects. This result justifies our choice of introducing 4 initial tokens as attention sinks in StreamingLLM.

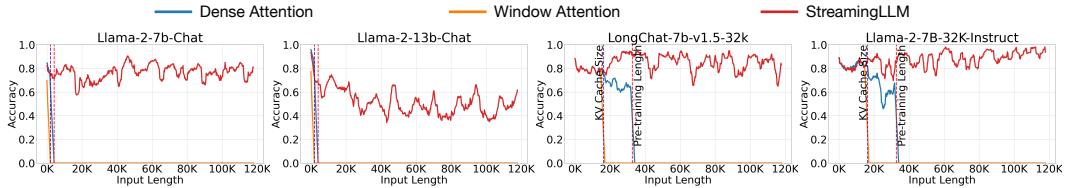
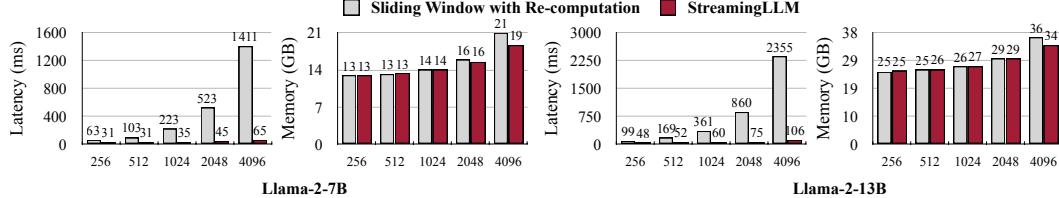


Figure 9: Performance on the StreamEval benchmark. Accuracies are averaged over 100 samples.

Figure 10: Comparison of per-token decoding latency and memory usage between the sliding window approach with re-computation baseline and StreamingLLM, plotted against the cache size (attention window size) on the X-axis. StreamingLLM delivers a remarkable speedup of up to  $22.2\times$  per token and retains a memory footprint similar to the re-computation baseline.

**Cache Sizes.** In Table 6, we evaluate cache size’s impact on StreamingLLM’s perplexity. Contrary to intuition, increasing the cache size doesn’t consistently lower the language modeling perplexity. This inconsistency shows a potential limitation where these models might not maximize the utility of the entire context they receive. Future research efforts should target enhancing these models’ capabilities to utilize extensive contexts better.

#### 4.5 EFFICIENCY RESULTS

We benchmark its decoding latency and memory usage against the sliding window with re-computation, which is the only baseline with acceptable performance. Both methods are implemented using the Huggingface Transformers library (Wolf et al., 2020) and tested on a single NVIDIA A6000 GPU using the Llama-2-7B and Llama-2-13B models. As depicted in Figure 10, as the cache size increases, StreamingLLM’s decoding speed demonstrates a linear growth. The sliding window with re-computation baseline has a quadratic rise in decoding latency. Thus, StreamingLLM achieves an impressive speedup, reaching up to  $22.2\times$  per token. Despite its reduced latency, StreamingLLM sustains a memory footprint consistent with the re-computation baseline.

## 5 CONCLUSION

Deploying LLMs in streaming applications is urgently needed but comes with challenges due to efficiency limitations and reduced performance with longer texts. Window attention provides a partial solution, but its performance plummets when initial tokens are excluded. Recognizing the role of these tokens as “attention sinks”, we introduced StreamingLLM—a simple and efficient framework that enables LLMs to handle unlimited texts without fine-tuning. By adding attention sinks with recent tokens, StreamingLLM can efficiently model texts of up to 4 million tokens. We further show that pre-training models with a dedicated sink token can improve the streaming performance. StreamingLLM firstly decouples the LLM’s pre-training window size and its actual text generation length, paving the way for the streaming deployment of LLMs.

Table 6: Effects of cache size on StreamingLLM’s performance. Increasing the cache size in StreamingLLM doesn’t consistently yield a decrease in perplexity across all models, suggesting these models may not be fully exploiting the provided context. Cache config  $x+y$  denotes adding  $x$  initial tokens with  $y$  recent tokens. Perplexity is evaluated on 400K tokens in the concatenated PG19 test set.

Cache	4+252	4+508	4+1020	4+2044
Falcon-7B	13.61	12.84	<b>12.34</b>	12.84
MPT-7B	<b>14.12</b>	14.25	14.33	14.99
Pythia-12B	13.17	12.52	<b>12.08</b>	12.09
Cache	4+508	4+1020	4+2044	4+4092
Llama-2-7B	9.73	9.32	<b>9.08</b>	9.59

## REFERENCES

- Ebtessam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. Falcon-40B: an open large language model with state-of-the-art performance. 2023.
- Sotiris Anagnostidis, Dario Pavlo, Luca Biggio, Lorenzo Noci, Aurelien Lucchi, and Thomas Hofmann. Dynamic context pruning for efficient and interpretable autoregressive transformers, 2023.
- Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020. arXiv:2004.05150.
- Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar van der Wal. Pythia: A suite for analyzing large language models across training and scaling, 2023.
- Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.
- bloc97. NTK-Aware Scaled RoPE allows LLaMA models to have extended (8k+) context size without any fine-tuning and minimal perplexity degradation., 2023. URL [https://www.reddit.com/r/LocalLLaMA/comments/14lz7j5/ntkaware\\_scaled\\_ropeAllows\\_llama\\_models\\_to\\_have/](https://www.reddit.com/r/LocalLLaMA/comments/14lz7j5/ntkaware_scaled_ropeAllows_llama_models_to_have/).
- Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Quantizable transformers: Removing outliers by helping attention heads do nothing, 2023.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Heben Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.
- Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation, 2023. arXiv: 2306.15595.
- Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%\* chatgpt quality, March 2023. URL <https://lmsys.org/blog/2023-03-30-vicuna/>.
- Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv:1803.05457v1*, 2018.
- Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. 2023.
- Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness, 2022. arXiv:2205.14135.

- Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.
- Tanya Goyal and Greg Durrett. Evaluating factuality in generation with dependency-level entailment. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Online, 2020. Association for Computational Linguistics.
- kaiokendev. Things I'm learning while training superhot., 2023. URL <https://kaiokendev.github.io/til#extending-context-to-8k>.
- Ehsan Kamalloo, Nouha Dziri, Charles L. A. Clarke, and Davood Rafiei. Evaluating open-domain question answering in the era of large language models, 2023.
- Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *8th International Conference on Learning Representations, ICLR 2020*. OpenReview.net, April 2020.
- Dacheng Li, Rulin Shao, Anze Xie, Ying Sheng, Lianmin Zheng, Joseph E. Gonzalez, Ion Stoica, Xuezhe Ma, , and Hao Zhang. How long can open-source llms truly promise on context length?, June 2023. URL <https://lmsys.org/blog/2023-06-29-longchat>.
- Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023.
- Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *EMNLP*, 2018.
- Evan Miller. Attention is off by one, 2023. URL <https://www.evanmiller.org/attention-is-off-by-one.html>.
- OpenAI. Gpt-4 technical report, 2023.
- Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1525–1534, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1144. URL <https://aclanthology.org/P16-1144>.
- Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models, 2023.
- Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*, 2022.
- Ofir Press, Noah Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations*, 2022. URL <https://openreview.net/forum?id=R8sQPPGCv0>.
- Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.
- Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*, 2020.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémie Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open foundation models for code, 2023.

- Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *arXiv preprint arXiv:1907.10641*, 2019.
- John Schulman, Barret Zoph, Christina Kim, Jacob Hilton, Jacob Menick, Jiayi Weng, Juan Felipe Ceron Uribe, Liam Fedus, Luke Metz, Michael Pokorny, et al. Chatgpt: Optimizing language models for dialogue. *OpenAI blog*, 2022.
- Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.
- Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. [https://github.com/tatsu-lab/stanford\\_alpaca](https://github.com/tatsu-lab/stanford_alpaca), 2023.
- MosaicML NLP Team. Introducing mpt-7b: A new standard for open-source, commercially usable llms, 2023. URL [www.mosaicml.com/blog/mpt-7b](http://www.mosaicml.com/blog/mpt-7b). Accessed: 2023-05-05.
- Together. Llama-2-7b-32k-instruct — and fine-tuning for llama-2 models with together api, June 2023. URL <https://together.ai/blog/llama-2-7b-32k-instruct>.
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.
- Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.
- Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. 2020.
- Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierrick Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface’s transformers: State-of-the-art natural language processing, 2020.
- Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.
- Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. In Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020*. Curran Associates, Inc., 2020.
- Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *CoRR*, abs/1905.07830, 2019. URL <http://arxiv.org/abs/1905.07830>.
- Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuhui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.
- Tianyi Zhang, Faisal Ladakh, Esin Durmus, Percy Liang, Kathleen McKeown, and Tatsunori B. Hashimoto. Benchmarking large language models for news summarization, 2023a.
- Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H<sub>2</sub>O: Heavy-hitter oracle for efficient generative inference of large language models, 2023b.

# Who Says Elephants Can't Run: Bringing Large Scale MoE Models into Cloud Scale Production

**Young Jin Kim**

Microsoft

youki@microsoft.com

**Rawn Henry**

NVIDIA

rhenry@nvidia.com

**Raffy Fahim**

Microsoft

raffybekheit@microsoft.com

**Hany Hassan Awadalla**

Microsoft

hanyh@microsoft.com

## Abstract

Mixture of Experts (MoE) models with conditional execution of sparsely activated layers have enabled training models with a much larger number of parameters. As a result, these models have achieved significantly better quality on various natural language processing tasks including machine translation. However, it remains challenging to deploy such models in real-life scenarios due to the large memory requirements and inefficient inference. In this work, we introduce a highly efficient inference framework with several optimization approaches to accelerate the computation of sparse models and cut down the memory consumption significantly. While we achieve up to 26x speed-up in terms of throughput, we also reduce the model size almost to one eighth of the original 32-bit float model by quantizing expert weights into 4-bit integers. As a result, we are able to deploy 136x larger models with 27% less cost and significantly better quality compared to the existing solutions. This enables a paradigm shift in deploying large scale multilingual MoE transformers models replacing the traditional practice of distilling teacher models into dozens of smaller models per language or task.

## 1 Introduction

Transformer models are getting larger and better on a continuous basis. The largest transformer models scale up to hundreds of billions of parameters, (Smith et al., 2022) resulting in high training and inference costs. This makes it difficult to deploy such models in any real-life scenario with reasonable latency and throughput. Mixture of Experts (MoE) models offer a more cost-effective method to scaling model sizes by using sparsely activated computations. More specifically, feed forward layers can be easily enlarged by replicating the original

weights  $E$  times where  $E$  is the number of experts. Each of these replicas is referred to as an expert, and tokens get routed to these experts depending on a gating function. Transformer models have a much larger number of parameters when utilizing these MoE layers. However, the number of flops remains comparable to their dense counterparts thanks to sub-linear scaling in computation costs (Shazeer et al., 2017). Recently, the Mixture of Experts (MoE) architecture has been successfully utilized to scale massive large scale multilingual models (Lepikhin et al., 2020)), NLU tasks (Fedus et al., 2021; Zoph et al., 2022) and multilingual multitask models (Kim et al., 2021).

MoE offers the benefits of scaling the model to gain better accuracy without paying the huge compute cost of massive dense models. However, large scale MoE models bring their own set of unique challenges to get efficient training and inference methods. Most of the previous work focused on improving training efficiency and throughput (Fedus et al., 2021; Kim et al., 2021). In this work, we focus on optimizing MoE models inference and latency since it is crucial to harvest the benefits of such models in real-life scenarios.

*Production-scale Multilingual Machine Translation systems:* in this work, we explore deploying MoE models for large scale Multilingual Machine Translation systems to benefit from large language models, while maintaining reasonable serving cost. Multilingual large scale systems are already very attractive due to multiple aspects. First, they benefit modeling since they allow better accuracy, especially through transfer learning across languages. Additionally, they improve deployment and serving since we can replace dozens of models with a single model that is able to serve many languages at the same time. Nevertheless, we need the infer-

ence to be highly optimized to make inference cost-efficient. Despite these benefits, shipping such multilingual models brings a new challenge, because they usually require a much larger model capacity in terms of the number of parameters and the computation. The MoE model architecture could be a promising solution given its sub-linear or constant FLOPs increase in terms of the number of model parameters. But, the large memory consumption issue still remains.

In this work, we show how to enable deploying a single MoE model that can serve many languages replacing dozens of traditional models while improving accuracy and maintaining latency, throughput and cost efficiency. We set the goal for this work to match latency and throughput of a distilled small model deployed on CPU while achieving better serving cost.

It is worth noting that while the optimizations presented here are applied to MoE encoder-decoder architecture for multilingual machine translation task, they are applicable to other architectures and tasks without any loss of generality. Given the recent success of MoE models on wide set of NLU and NLG tasks (Fedus et al., 2021; Zoph et al., 2022), we believe the optimization presented in this work will be equally enabling to other tasks as it is for machine translation.

## 2 Challenges and Contributions

### 2.1 MoE Inference challenge

Even though the MoE architecture in theory requires much less computation with larger number of parameters, it adds several computations such as token routing and all-to-all communication which could be a significant hit to the training throughput as much as 12% for a single node as shown in (Liu et al., 2022). In addition, it significantly increases the amount of memory traffic in the MoE layers. So far, previous studies focused more on the training efficiency of those MoE models and there has not been a solution to deploy this kind of models into the real-time applications. At inference time, we have observed the naive implementation of MoE models could be up to 30 times slower than its dense counterpart with the same embedding and hidden dimensions. To achieve a reasonable deployment cost, it is critical to lower the inference cost by increasing throughput and reducing the latency. Since MoE layers are not widely optimized for the inference scenarios, it is challenging to build

efficient runtime environment in terms of computation and memory consumption.

Recently, (Rajbhandari et al., 2022) introduced several approaches to improve inference of MoE models focusing on very large scale models larger than 100B parameters and decoding on multiple GPUs. When the model size increases beyond the memory limit of a single GPU, multiple GPUs can be used together for a single inference by splitting the model weights across different GPUs. While multi-gpu can reduce latency and is required to serve extremely large models, it introduces significant communication overhead and makes it more difficult to scale up and down the number of instances based on traffic. Therefore, even though multiple GPUs could bring much larger models into production, we focus on the single GPU inference scenario due to its cost efficiency with reasonably sized models. It is worth noting that the optimization we are presenting here for single GPU can be utilized for larger models on several GPUs as well. However, this is beyond the scope of this paper.

### 2.2 Inference Optimization Contributions

In this paper, we show how to reduce the memory requirements to deploy largest possible model on a single GPU, which avoids costly all-to-all collectives. In addition, we optimized routing efficiency for GPUs and implemented batch pruning. We describe how we extend NVIDIA’s FasterTransformer<sup>1</sup> inference framework to support the MoE model architecture in a real world deployment scenario:

- We present how we utilize the parallel primitives in the CUTLASS<sup>2</sup> and CUB<sup>3</sup> libraries to efficiently express token routing and the batched matrix multiply required for MoE.
- We propose a new GEMM (GEneral Matrix Multiply) which can consume 4-bit/8-bit quantized weights and perform float math. The new GEMM works as drop-in replacements of normal feedforward layers without having additional logic to handle quantization/dequantization of activations. We also show that 4/8 bit weight-only quantization preserves the accuracy without any additional algorithms.

<sup>1</sup><https://github.com/NVIDIA/FasterTransformer>

<sup>2</sup><https://github.com/NVIDIA/cutlass>

<sup>3</sup><https://github.com/NVIDIA/cub>

- We implement an effective batch pruning algorithm for MoE layers to make the search algorithm on the decoder very efficient.

### 2.3 FasterTransformer overview

We build our MoE optimization over NVIDIA’s FasterTransformer, a highly optimized open source inference engine for transformer models. FasterTransformer implements a highly optimized transformer layers for both the encoder and decoder for inference which is built on top of CUDA, cuBLAS, cuBLASLt and C++. FasterTransformer supports seamless integration with Triton Inference server<sup>4</sup> which enabled us to deploy our models in scalable large scale cloud environment.

We have extended FasterTransformer to support DeepSpeed MoE models(Kim et al., 2021) and added support for Transformer with Untied Positional Encoding (TUPE) (Ke et al., 2020) attention, gate routing and efficient computation of MoE layers, including batch pruning in those layers.

## 3 MoE Inference Optimizations

### 3.1 Model architecture

MoE showed tremendous success with encoder-decoder model architecture in Multilingual Machine Translation (Lepikhin et al., 2020; Kim et al., 2021), and in Natural Language understanding (Fedus et al., 2021; Zoph et al., 2022). Therefore, in this work we focus on the encoder-decoder architecture without loss of generality since the optimization is directly applicable to encoder-only and decoder-only models as well.

We train an encoder-decoder model for machine translation with deep encoder and shallow decoder architecture as proposed in (Kim et al., 2019; Kasai et al., 2020). For a given batch of input sentences, the encoder is executed only once while the decoder is executed multiple times with a beam search algorithm per token. The auto-regressive execution of the decoder is usually the performance bottleneck. Therefore, utilizing a shallow decoder partially mitigates that effect. Empirically, we have found that using half number of decoder layers than the number of encoder layers gives a good trade-off between quality and performance. For the most efficient MoE layer execution, we use top-1 gating algorithm proposed in Switch transformers (Fedus et al., 2021). At every other layer, MoE layer is used instead of the plain feedforward layer.

We use embedding dimension of 1024, the positional and word correlations are computed separately and added together in the self attention module (TUPE) (Ke et al., 2020). The feed-forward hidden dimension is 4096 with 24 encoder layers and 12 decoder layers as proposed in (Kim et al., 2021). This model configuration satisfies the deep encoder and shallow decoder design and the model weights fit well into the GPU memory without tensor slicing model parallelism (Shazeer et al., 2018). The tensor slicing approach increases communication overheads and could potentially introduce training instability issues. In the production setting, we choose a model building pipeline which could minimize such instability. On the other hand, expert parallelism is preferred over tensor slicing model parallelism because an atomic layer operation such as a feedforward layer is executed inside one GPU. Therefore, we increase the number of model parameters by adding more experts. With the size of the layers and the number of layers, the total number of parameters is roughly 5 billion when 32 experts are used in the MoE layers. With half precision floating point (fp16), this is about 10 GB which can fit on a single 16 GB GPU.

### 3.2 Multilingual Machine Translation Model

The traditional Machine Translation deployment paradigm generally follows the teacher-student model. Where several teachers are being distilled into a very small student model that get deployed on CPU (Kim et al., 2019). For instance, deploying 100 languages translation system, would require training, distilling and deploying at least 200 of such models. Each model is trained individually for a particular language pair. This is not scalable since each individual model needs to go through various model compression steps to be deployed on CPUs with relatively low FLOPs numbers. This not only hinders scalable model building, but also knowledge sharing and transfer between different language pairs and tasks. Multilingual training approaches have been utilized to overcome this problem. However, shipping these multilingual models brings a new challenge since such models usually require much larger capacity in terms of the number of parameters and the computation.

In this work, we use a multilingual MT system trained on 10 language pairs and can be used in place of individual systems per language pair. The model is trained using production scale training

---

<sup>4</sup><https://github.com/triton-inference-server/server>

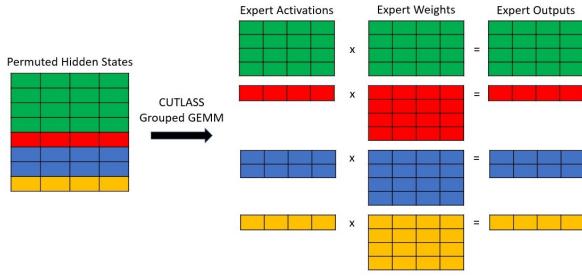


Figure 1: Shows the computation performed by CUTLASS Grouped GEMM. Each color is a sub-matrix for a particular expert, with the matrix multiplies for each expert happening in parallel. If the yellow sentence was finished, it would be omitted from the computation with batch-pruning enabled. This would completely remove the need to load the weight matrix for the yellow expert.

data of up to  $\sim 4B$  training sentence pairs with a vocabulary of 128K using Sentence Piece<sup>5</sup>

### 3.3 Optimized GPU kernel design

One key factor to get an optimal performance with massive CUDA cores is to have efficient parallel algorithms for various additional operations for MoE. In MoE layers, each row in the input activation must get routed to a specific expert weight matrix, depending on a **top- $k$  gating function**. We implement this routing as a GPU friendly radix sort using NVIDIA’s highly efficient CUB library.

In this case, each row in the activation matrix is a token to be translated. The top- $k$  gating function outputs a list with  $k$  (*expert\_scale*, *expert\_idx*) tuples for each input token. Thus, for top-1 gating (as is done in our case), the function outputs a single tuple for every row of the activation matrix.

In order to perform the routing, we first append the index for each row to the end of the tuple giving a tuple of (*expert\_scale*, *expert\_idx*, *row\_idx*). Then, we sort the tuple using *expert\_idx* as the keys in order to group all rows that will be processed by the same *expert\_idx* together. The *row\_idx* entry from the sorted tuples are then used to permute the original activation matrix in global memory to a layout where all rows routed to the same expert are laid out contiguously in memory.

In order to finalize the routing, we view each group of rows assigned to a particular expert as its own sub-matrix and compute pointers to the start of these sub-matrices. We then pair each sub-matrix pointer with pointers to the weights and biases for

the expert they are routed to, and use CUTLASS Grouped GEMM to compute all of these matrix multiplies in parallel using a single kernel. Figure 1 shows the computation performed by CUTLASS.

Finally, we un-permute the rows to their original ordering and apply the *expert\_scale* to each row before passing the output of the MoE module to the other parts of the network.

### 3.4 Expert quantization with 4-bit and 8-bit

We quantize the MoE weights for two reasons:

1. MoE weights are extremely large which limits the size of the models that can fit on the common 16 GB inference cards such as T4.
2. MoE matrix multiplies require loading the weights for several different experts which results in them being memory bound.

We do not use Quantization Aware Training (QAT) (Wu et al., 2020), because our quantization approach does not degrade model performance. QAT is usually used when there exists a noticeable performance degradation from quantization. Also, we focus on quantizing expert **weights only**, because they are contributing to more than 90% of entire model weights thanks to the special property of MoE model size scaling. We get much larger model mostly from the expert parameters in MoE layers (Shazeer et al., 2017).

---

#### Algorithm 1: Weight dequantize

---

```

Input : E - Number of Experts
Input : W - quantized weights of shape (E, M, N)
Input : S - FP16 scales of shape (E, 1, N)
Output : FP16 dequantized weights

1  $W_{dq} \leftarrow \text{NewMatrix}(E, M, N)$ 
2 for  $e \leftarrow 0$  to  $E - 1$  do
3   for  $m \leftarrow 0$  to  $M - 1$  do
4     for  $n \leftarrow 0$  to  $N - 1$  do
5        $f = \text{IntToFloat}(W[e, m, n])$ 
6        $W_{dq}[e, m, n] = f * S[e, n]$ 
7     end for
8   end for
9 end for
10 return  $W_{dq}$ 

```

---

All **activations and biases are kept as FP16** and only the expert weight matrices are quantized. As a result, we do not require any post-training calibration (because we don’t need scales for the activations) which makes this recipe easy to apply to several language families. We perform **symmetric, range-based per-channel quantization** on each expert weight. This means that for expert weights of

<sup>5</sup><https://github.com/google/sentencepiece>

shape  $(E, M, N)$  where  $E$  is the number of experts and  $M$  and  $N$  are arbitrary dimensions, we produce scales of shape  $(E, 1, N)$ . The same quantization method is used for `int4` and `int8`. During inference, we dequantize the weights to FP16 and perform our matrix multiplies using floating point computations. Algorithm 1 shows the dequantization performed during inference.

One option for implementing the GEMM + Dequantize would be to write a separate kernel to dequantize the weights before the MoE GEMM. However, this would actually increase the amount of memory traffic as we would **add a read of  $W$  and a write to  $W_{dq}$**  as shown in Algorithm 1. As a result, we decided to take advantage of the flexibility of CUTLASS and **fuse the dequantize step into the GEMM kernel**. After profiling, we realized that **the conversion from int to float (line 5 in Algorithm 1)** was slower than anticipated. In order to improve this, we replaced the native int to float conversion (I2F) with a series of **high throughput ALU and FP16 instructions** which improved the performance of our fused GEMM + Dequantize.

### 3.4.1 Quantization Optimization

The conversion optimization mentioned above produces exact results to the native I2F conversions. It relies on two key observations:

FP16中[1024,2048]中的数值，在位数上的分布规律，比如1024刚好是 $2^{10}$ ，而指位数表达的下一个值上 $2^{11}=2048$ ， $2^{10}$ 和 $2^{11}$ 的跨度是1024，(1024,2048)间的值无法再通过指位数表达。只能通过位数来表达。即 For any FP16 number  $X$  where  $1024 \leq X <$  位数部分，是直接存储的。int( $x - 1024$ )的一进制，方向为2048，1024 will be represented exactly in the int( $x - 1024$ )的一进制，方向为2048，1024 will be represented exactly in the 由低位至高位。比如3.....因此，扩展开来，对于 exponent bits and int( $X - 1024$ ) will be divided into mantissa bits of its representation.至于NV为啥选的是1024而不是512，我也不懂。但有一点从理解，就是FP16的尾数为10位，能够精确表达的数值最大是1024。比如2048~4096，10位二进制是无法精确表达[1,2048]的。 $(1.\text{尾数}) \cdot 2^{11}$ ，尾数部分能表达的最小单位为 $(1/2)^{10} \cdot 2^{11-2} = 1/2^5$ ，不是1/2。For any integer  $0 \leq Y < 1024$ , we can construct the FP16 representation of  $Y + 1024$  by setting the exponent to 1024 and storing  $Y$  保存在尾数部分。所以它必须就是1024。

为 $[0,1024)$ 的int数值，构造一个FP16表达式，将int的二进制直接保存在尾数部分即可。此时 $Y' = Y + 1024$ ；也就是此时的16位二进制，表达的是 $Y \times 1024$ 。为了获得FP16表达式，我们也很简单，就是用 $Y \times 1024$ 量化 the weights，我们 add 128 to int8 weights and 8 to int4 weights to make them all unsigned. 其实设计得非常巧妙，因为这样就避开了使用右位。很低的cv1指令做数值运算，而是变成使用高位操作和数值运算指令。Our optimization exploits these observations to quickly convert int4s or int8s and FP16. After we quantize the weights, we add 128 to int8 weights and 8 to int4 weights to make them all unsigned. We refer to these weights as  $W_+$ . This is not strictly necessary, but removes the need to perform sign extension logic.

### 3.4.2 Optimized 8-bit Dequantize

In order to best utilize the hardware, we convert int8s to FP16s two at a time, leveraging the fact that 2 FP16 elements can fit in a 32-bit register. This is done as follows:

1. We load 4 int8 values,  $[e_0, e_1, e_2, e_3]$  from  $W_+$  into a single 32-bit register.
  2. We then create a second 32-bit register,  $R_1$ , that stores the FP16 representation of  $[e_0 + 1024, e_1 + 1024]$  leveraging observation (2).
  3. Next, we use float math to subtract  $[1152, 1152]$  from  $R_1$ . This subtraction is due to the fact that we must subtract 1024 from each number in  $R_1$  convert  $e_0$  and  $e_1$  to FP16. Then, we must further subtract 128 from each number to obtain the float representation of the original, signed integer.
  4. Lastly, we repeat steps 2 and 3 for  $e_2$  and  $e_3$ .

### 3.4.3 Optimized 4-bit Dequantize

We change the layout of the weights to reduce the number of logic instructions needed to construct the FP16s  $[e_i + 1024, e_{i+1} + 1024]$ . Thus, for int4, we change the layout of  $W_+$  to reorder groups of 8 elements as follows:

$$[e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7] \rightarrow [e_0, e_2, e_4, e_6, e_1, e_3, e_5, e_7]$$

With this new layout, the idea for int4 is similar to what was previously described for int8. Of course, we must now subtract [1032, 1032] to recover the original, signed integer as fp16. We must also iterate 4 times since 1 32-bit register holds 8 int4s and conversion happens 2 at a time.

### 3.5 MoE Batch Pruning

Batch pruning refers to the act of removing sentences from a batch dynamically as soon as they are done translating. We observed that this speeds up MoE layers as it can prevent the loading of entire expert weights, reducing the amount of memory traffic required in these memory bound layers.

In order to implement batch pruning in the MoE layers, we make a simple modification to the gating function so that it assigns a large expert\_idx to all finished sentences. This causes all finished sentences to be moved to the end of the permuted activation matrix in the routing step. To complete

Table 1: Throughput of quantized MoE GEMMs normalized against the throughput of the FP16 MoE Gemm. The number of active experts is the number of experts that receive tokens from routing. The matrix shapes for the GEMM  $C = A @ B$  are  $A=mx1024$ ,  $B=1024x4096$ , where  $m$  is different for each expert. The total number of tokens is set to 40 since this is close to what the decoder computes in our inference environment.

Active Experts	FP16	Int8 native I2F	Int8 optimized I2F	Int4 optimized I2F
1	1	1.05	1.28	1.24
4	1	1.01	1.21	1.28
8	1	1.34	1.21	1.57
16	1	1.40	1.39	1.73
24	1	1.40	1.49	1.78
32	1	1.46	1.59	1.85
GEOMEAN	1	1.26	1.35	1.56

the pruning, we simply keep track of the total number of active tokens and only process the first active\_tokens rows of the permuted activation matrix mentioned in section 3.3.

## 4 Results and discussion

All experiments in this section are run on a single NVIDIA PCIE V100 running inside a docker container running Ubuntu 20.04 and CUDA 11.6. All code is compiled with nvcc and gcc/g++ 9.3.

We run our experiments considering an encoder-decoder MoE model with 32 experts with TUPE (Ke et al., 2020), similar to the setup in (Kim et al., 2021) but with a vocabulary size of 128k. All throughput metrics measure the time to translate 1000 tokenized English sentences ( $\sim 40$ K tokens) to German (en-de) or vice-versa (de-en) and record the total number of input tokens translated per second. BLEU metrics are reported on the same data set.

### 4.1 Speed-up and Cost-Effectiveness

We measure the improvement of our batch pruning optimization by comparing the throughput with and without that optimization. We found that we achieve up to  $1.14\times$  speed up relative to our optimized baseline without batch pruning.

**INT8/INT4 GEMM Performance.** First, Table 1 shows a performance comparison for the FP16 GEMM compared to fused GEMM + Dequantize with native I2F and our optimized I2F sequence for INT8. Our INT4 implementation only supports the optimized I2F sequence. Depending on the number of experts, INT8 and INT4 could accelerate MoE computation up to 59% and 85%, respectively.

**INT8/INT4 Quality Impact.** We also consider the impact of INT8 and INT4 expert quantization on BLEU scores, we observe negligible translation quality degradation when quantizing model weights. Table 2 shows the change in BLEU compared to FP16 after applying quantization.

**End-to-end Performance Improvements.** Table 3 shows our machine translation experiments for EN-DE, with different batch sizes and different quantization schemes and reports both the throughput of our PyTorch and Faster Transformer implementations. Compared to the Torch-FP16 baselines, the optimizations applied achieve significant speed-up across different settings.

**Cost Comparison.** Table 4 shows the deployment cost comparison between the MoE models and smaller models optimized for CPU deployment (Kim et al., 2019). The cost of deploying MoE models which are 136x larger on CPU is more than 100 times of the cost of deploying smaller models on CPU. However, the optimized large MoE models on GPU cost less than the current CPU model deployment with smaller models.

Table 2: BLEU differences from INT8 and INT4 weight-only compared to the FP16 baseline.

Language Pair	Beam 1 $\Delta$ BLEU	
	INT8	INT4
EN-DE (Beam 1)	-0.028	-0.052
EN-DE (Beam 2)	0.051	-0.180
DE-EN (Beam 1)	-0.084	0.044
DE-EN (Beam 2)	-0.027	-0.031
Avg. of 10 language pairs (Beam 2)	-0.007	-0.167

## 5 Conclusions and Future Work

This paper describes how to make large MoE models cost-efficient on a single GPU in a real-world inference environment. The final implementation achieves a speedup of up to 26X over PyTorch baseline. Our GPU MoE implementation allows serving much larger and higher-quality models compared to dense models on CPUs without increasing the cost of serving. We consider two main avenues for future work. We are currently working on improving our fused GEMM + Dequantize kernel to enable the use of fully vectorized 16 byte loads on the weight matrix. In addition, we plan to explore deploying even larger models with distributed inference in the future in a cost-efficient way.

Table 3: Throughputs for beam=1 and beam=2 for varying batch sizes. Throughput is measured as input tokens processed per second. The precisions (FT-INT8 and FT-INT4) in the table refer to the quantization applied to the MoE weights. *Torch-FP16* columns show the throughput numbers when we run the model with PyTorch v1.10 using FP16 model weights.

Batch Size	Beam=1 Input tokens processed/sec				Beam=2 Input tokens processed/sec			
	Torch-FP16	FT-FP16	FT-INT8	FT-INT4	Torch-FP16	FT-FP16	FT-INT8	FT-INT4
1	16	388	401	400	14	351	361	361
8	70	1594	1639	1662	65	1453	1507	1518
20	150	3025	3178	3247	139	2571	2719	2803
32	214	4008	4264	4379	202	2960	3137	3239
64	379	5371	5706	5935	349	4333	4578	4746
96	485	6689	7101	7483	440	5062	5384	5605

Table 4: Deployment cost comparison. We show the most cost-effective throughputs under our 1s latency budget.

Hardware	Parameters	Batch size	Price (East US)	Latency (ms)	Throughput (words/sec)	Monthly USD/token
CPU (AVX512)	0.04 B	1	\$587.65 (F16s)	75	351	0.209
CPU (AVX512)	5.32 B	1	\$587.65 (F16s)	1080	26	22.602
NVIDIA T4	5.32 B	20	\$390.55 (NC4as T4 v3)	421	1565	0.250
NVIDIA T4	5.32 B	64	\$390.55 (NC4as T4 v3)	824	2560	0.153

## Ethics Statement

The authors have put the best effort to comply with the [ACL Ethics Policy](#). For the experiments, we have used WMT public domain datasets and respected the license policy for our usage.

## Acknowledgements

We thank the Microsoft Z-Code team and the Microsoft Translator team for the great effort to push the limit of the production quality in machine translation and to quickly adopt the state-of-the-art Mixture of Experts models into the cloud-scale production. We also thank the Microsoft DeepSpeed team for the collaboration on more efficient and scalable Mixture of Experts architecture and library development. Additionally, we are deeply grateful for the amazing work of the NVIDIA CUTLASS team which developed grouped GEMM kernels which were crucial to get good performance with Mixture of Experts. We also thank the CUTLASS team for answering all of our questions to help us implement efficient kernels that handle GEMMs with different input types. Lastly, we thank the NVIDIA Faster-Transformer team for their help with integrating our Mixture of Experts implementation into their efficient transformer inference framework.

## References

- William Fedus, Barret Zoph, and Noam Shazeer. 2021. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity.
- Jungo Kasai, Nikolaos Pappas, Hao Peng, James Cross, and Noah A Smith. 2020. Deep encoder, shallow decoder: Reevaluating non-autoregressive machine translation. *arXiv preprint arXiv:2006.10369*.
- Guolin Ke, Di He, and Tie-Yan Liu. 2020. Rethinking positional encoding in language pre-training. *CoRR*, abs/2006.15595.
- Young Jin Kim, Ammar Ahmad Awan, Alexandre Muzio, Andres Felipe Cruz Salinas, Liyang Lu, Amr Hendy, Samyam Rajbhandari, Yuxiong He, and Hany Hassan Awadalla. 2021. Scalable and efficient moe training for multitask multilingual models. *arXiv preprint arXiv:2109.10465*.
- Young Jin Kim, Marcin Junczys-Dowmunt, Hany Hassan, Alham Fikri Aji, Kenneth Heafield, Roman Grundkiewicz, and Nikolay Bogoychev. 2019. From research to production and back: Ludicrously fast neural machine translation. In *Proceedings of the 3rd Workshop on Neural Generation and Translation*, pages 280–288.
- Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020.

Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*.

Rui Liu, Young Jin Kim, Alexandre Muzio, and Hany Hassan. 2022. Gating dropout: Communication-efficient regularization for sparsely activated transformers. In *International Conference on Machine Learning*, pages 13782–13792. PMLR.

Samyam Rajbhandari, Conglong Li, Zhewei Yao, Min-jia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale.

Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*.

Shaden Smith, Mostafa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. 2022. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model.

Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. 2020. Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602*.

Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. 2022. St-moe: Designing stable and transferable sparse expert models.

# ZeroQuant: Efficient and Affordable Post-Training Quantization for Large-Scale Transformers

Zhewei Yao\*, Reza Yazdani Aminabadi, Minjia Zhang  
 Xiaoxia Wu, Conglong Li, Yuxiong He

Microsoft

{zheweyao, yazdani.reza, minjiaz, xiaoxiawu, conglong.li, yuxhe}@microsoft.com

## Abstract

How to efficiently serve ever-larger trained natural language models in practice has become exceptionally challenging even for powerful cloud servers due to their prohibitive memory/computation requirements. In this work, we present an efficient and affordable post-training quantization approach to compress large Transformer-based models, termed as ZeroQuant. ZeroQuant is an end-to-end quantization and inference pipeline with three main components: (1) a fine-grained hardware-friendly quantization scheme for both weight and activations; (2) a novel affordable layer-by-layer knowledge distillation algorithm (LKD) even without the access to the original training data; (3) a highly-optimized quantization system backend support to remove the quantization/dequantization overhead. As such, we are able to show that: (1) ZeroQuant can reduce the precision for weights and activations to INT8 in a cost-free way for both BERT and GPT-3-style models with minimal accuracy impact, which leads to up to 5.19x/4.16x speedup on those models compared to FP16 inference; (2) ZeroQuant plus LKD affordably quantize the weights in the fully-connected module to INT4 along with INT8 weights in the attention module and INT8 activations, resulting in 3x memory footprint reduction compared to the FP16 model; (3) ZeroQuant can be directly applied to two of the largest open-sourced language models, including GPT-J<sub>6B</sub> and GPT-NeoX<sub>20B</sub>, for which our INT8 model achieves similar accuracy as the FP16 model but achieves up to 5.2x better efficiency.

## 1 Introduction

Large-scale natural language models have been widely adopted in different applications, e.g., natural language understanding using BERT [63] and generation tasks using GPT-style models [48]. Although those models have achieved cutting-edge accuracy results, as the model size keeps increasing dramatically, the requirements of memory footprint and the computational cost to deploy them become a major bottleneck, even on cloud servers with powerful GPU devices.

One promising way to alleviate this challenge is quantization, which can reduce the bit precision for both weight and activations for lower memory footprint and faster compute (e.g., INT8 Tensor cores on T4/A100). However, quantization usually requires retraining (also known as quantization aware training, or QAT in short) to recover the accuracy degradation from representation loss of weight and activations. To enable QAT, the full training pipeline is usually required, including the training data and compute resources, to finetune the model. Access to those components is now oftentimes not available, and QAT is also a time-consuming process, particularly for those large-scale models.

Recently, zero-shot quantization [9, 46] and post-training quantization (PTQ) [45, 38] are proposed to address the training-data access and compute requirement challenges since PTQ generally requires no (or minimal) retraining. But most of those works primarily focus on computer vision problems on relatively

---

\*Code will be released soon as a part of <https://github.com/microsoft/DeepSpeed>

small scales. More recently, [6] shows promising PTQ results on BERT. However, (1) its main focus is on high-precision quantization (INT8/FP16) on  $\text{BERT}_{\text{base}}$ , (2) it does not consider other billion-scale generative models (GPT-3-style models [8]). More importantly, most of these works do not report real latency improvement, putting the usefulness of these methods in improving inference latency into question. For example, existing work often do not discuss the quantization/dequantization cost associated with different quantization schemes, which in fact has a big impact to the performance benefit of using low precision.

Besides, for extreme quantization (e.g., INT4), knowledge distillation is usually used to boost performance, which adds another source of expensive computation cost as compared to QAT. Furthermore, in order to achieve better accuracy performance, hidden-states knowledge distillation, e.g., [2, 79], is usually applied for the quantized model. This would put significant pressure on the GPU memory and the compute resource requirement since both the teacher and student models needed to be loaded into the GPU memory for training.

In this paper, we present ZeroQuant, an end-to-end post-training quantization and inference pipeline, to address those challenges, targeting both INT8 and INT4/INT8 mixed-precision quantization. Specifically, our contributions are:

- We apply fine-grained hardware-friendly quantization schemes on both weight and activations, i.e., group-wise quantization for weight and token-wise quantization for activations. Both quantization schemes can significantly reduce the quantization error and retain hardware acceleration properties.
- We propose a novel layer-by-layer knowledge distillation method (LKD) for INT4/INT8 mixed-precision quantization, where the neural network is quantized layer-by-layer through distillation with minimal iterations and even without the access to the original training data. As such, at any given moment, the device memory is primarily populated only with a single extra layer’s footprint, making billion-scale model distillation feasible with limited training budget and GPU devices.
- We develop a highly optimized inference backend, which eliminates the expensive computation cost of quantization/dequantization operators, enabling latency speedups on INT8 Tensor cores on modern GPU hardware.
- Our empirical results show that:
  - ZeroQuant enables quantizing BERT and GPT-3-style models into INT8 weight and activations to retain accuracy without incurring any retraining cost. Compared to FP16 inference, our INT8 model achieves up to 5.19x/4.16x speedup on  $\text{BERT}_{\text{base}}/\text{GPT-3}_{350M}$  on A100 GPUs.
  - ZeroQuant plus LKD can do INT4/INT8 mixed-precision quantization for BERT and GPT-3-style models. This results in a 3x memory footprint reduction with marginal accuracy loss as compared to the FP16 model. Also, thanks to the lightweight of LKD, we can finish the quantization process in 33s (10 minutes) for  $\text{BERT}_{\text{base}}$  ( $\text{BERT}_{\text{large}}$ ). We also demonstrate that LKD can use other datasets to achieve similar performance to the original training data.
  - We demonstrate the scalability of ZeroQuant on two of the largest open-sourced language models, i.e., GPT-J<sub>6B</sub> and GPT-NeoX<sub>20B</sub>, with INT8 quantization. ZeroQuant can achieve 3.67x speedup over the FP16 model for GPT-J<sub>6B</sub> and (2) reduce the GPU requirement for inference from 2 to 1 and latency from 65ms to 25ms for GPT-NeoX<sub>20B</sub> (i.e., 5.2x better system efficiency in total).

## 2 Related Work

Model compression has been explored from different aspects [25, 37, 39, 34, 43, 20, 24, 50, 18, 74, 40, 26, 55, 59, 28, 60, 68, 33, 14, 38, 31]. Among those, quantization is one of the most promising directions as it directly reduces the memory footprint and compute intensity. Here, we focus on quantization for NLP models and briefly discuss the related work.

The majority of quantization works can be categorized into quantization-aware training (QAT). [56, 76] are the first few works to quantize BERT models using integer numbers for both weight and activations. Particularly, [56] utilizes Hessian information to push the weight bit-precision to even INT2/INT4, and it also proposes group-wise quantization to quantize the weight matrix in a more fine-grained granularity compared to single matrix quantization. [21] introduces quantization noise to alleviate the variations of QAT. [79, 2]

leverage very expensive knowledge distillation [26] and data augmentation [28] to ternarize/binarize weights. [29] combines knowledge distillation [28] and learned step size quantization [19] to quantize the weight to 2–8 bits. Recently, [61] also uses knowledge distillation to compress GPT-2 models on task-specific problems to INT2. All those works quantize models using the original training datasets. More importantly they need retraining or finetuning the full model to recover the accuracy, and such compute cost on extra-large models, like [57, 11], can be hardly affordable for most research labs or practitioners.

One solution to overcome the compute cost challenge is post-training quantization (PTQ). However, PTQ often induces a significant drop in accuracy because the network can be sensitive to quantization errors. Along this line, one of the first works applied to Transformer-based [64] models is [75]. The authors introduce centroid-based quantization method, where outlier numbers use FP32 format and the rest numbers are quantized using non-uniform quantization. As such, it is hard to get the real inference latency benefit on general compute accelerators, e.g., CPU and GPU, because the parallel processing units in these hardware do not support efficient computation of mixed data types. More recently, [6] introduces high-precision activation quantization (FP16) for part of the model to overcome the high dynamic activation ranges. However, to the best of our knowledge, (1) How to apply PTQ on GPT-3-style models while achieving high accuracy has not been studied in any of previous work yet; (2) How to apply PTQ on billion (or even a dozen of billions) scale model is still under-explored; (3) Efficient inference system backend is still missing, especially for fine-grained quantization schemes, making it hard to achieve low latency on commodity hardware. ZeroQuant resolves all those limitations by considering the system backend into the algorithm design and we verify its capability on both BERT and large-scale GPT-3-style (up to 20 billion, i.e., GPT-NeoX<sub>20B</sub>) models for various tasks.

### 3 Background and Challenge

We give a brief overview of the transformer architecture and quantization background in Appendix A. Please refer to [64] and [23] for more details about the transformer architecture and quantization.

Post-training quantization (PTQ) exhibits great compression efficiency compared to quantization-aware training (QAT) since PTQ is usually applied to quantize the model without retraining. A common strategy of PTQ is to feed the training data to the network and calibrate the scaling factor,  $S$ , using the running mean. Please see Appendix B.1 for more details.

Some work has been done for BERT<sub>base</sub> models [6] with INT8 weight and mixed INT8/FP16 activation quantization. However, there is no investigation for (1) even lower bit-precision PTQ on BERT models and (2) large-scale GPT-3-style models. Here, we briefly discuss the challenge of the application of PTQ on both BERT (in Appendix C) and GPT-3-style models.

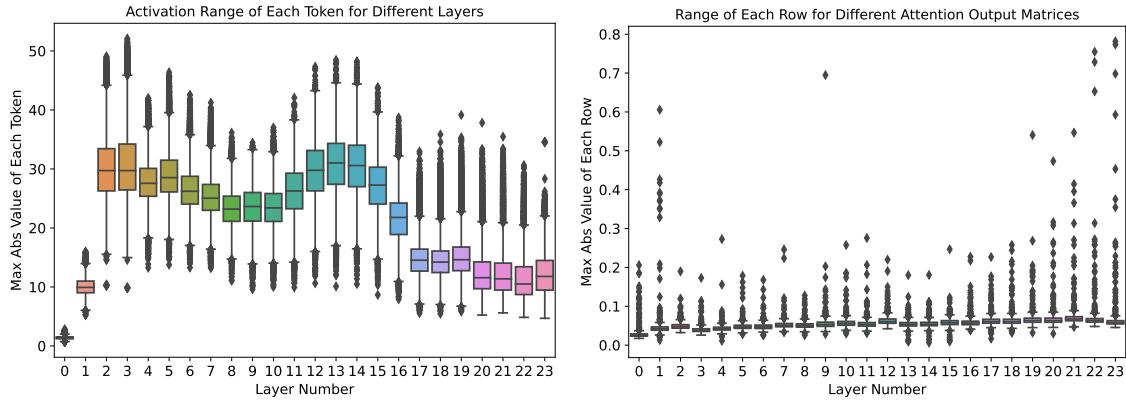


Figure 1: The activation range (left) and row-wise weight range of the attention output matrix (right) of different layers on the pretrained GPT-3<sub>350M</sub>. See Figure C.1 for the results of BERT<sub>base</sub>.

The results of GPT-3<sub>350M</sub> with PTQ are shown in Table 1. As can be seen, the INT8 activation

Table 1: Post training quantization results of GPT-3<sub>350M</sub> on 20 zero-shot evaluation datasets. Here WxAy means x-/y-bit for weight/activation. Particularly, for W4/8, we quantize the MHSA’s weight to INT8 and FFC’s weight to INT4. Please see Table H.1 for the results of all 20 tasks.

Precision	Lambada (↑)	PIQA (↑)	OpenBookQA (↑)	RTE (↑)	ReCoRd (↑)	Ave. 19 Tasks (↑)	Wikitext-2 (↓)
W16A16	49.3	66.3	29.4	53.8	75.1	38.9	21.5
W8A16	49.3	66.1	29.6	54.2	74.8	38.5	22.1
W16A8	44.7	64.8	28.2	52.7	69.2	37.8	24.6
W8A8	42.6	64.1	28.0	53.1	67.5	37.8	26.2
W4/8A16	0.00	51.4	30.2	52.7	16.1	28.9	1.76e5

quantization (i.e., the row of W16A8) causes the primary accuracy loss. Further pushing the weight to INT8 (i.e., the row of W8A8) does not change the accuracy of zero-shot evaluation tasks but leads the causal language modeling task (Wikitext-2) to worse perplexity score, which demonstrates the sensitivity of generation tasks as compared to other zero-shot evaluation problems. For W4/8A16, on some accuracy-based tasks, GPT-3<sub>350M</sub> still achieves reasonable performance like OpenBookQA but it loses accuracy on the majority of the rest tasks. Particularly, for Wikitext-2, GPT-3<sub>350M</sub> with W4/8A16 cannot generate any meaningful text anymore. Please also see Appendix C for the analysis for BERT.

**Dynamic Activation Range** To investigate why INT8 activation leads to significant accuracy drop for both BERT and GPT-3-style models, we plot the token-wise (i.e., the hidden state of each token) range of each activation for different transformer layers of GPT-3<sub>350M</sub> in Figure 1 (left). As can be seen, different tokens have dramatically different activation ranges. For example, the maximum range of the last layer is around 35 but the minimum range is close to 8. This larger variance in the activation range makes it difficult to use a fixed quantization range (usually the maximum value) for all tokens to retain the prediction accuracy, because the limited representation power for small range tokens is going to hurt the accuracy performance. **Different Ranges of Neurons in Weight Matrices** Similarly, we plot the row-wise (i.e., the output dimension) weight range of the attention output matrix ( $\mathbf{W}_o$ ) of GPT-3<sub>350M</sub> in Figure 1 (right). There is a 10x difference between the largest magnitudes of different rows and this leads to the worse generation performance of the INT8 weight PTQ. This also makes it very challenging when INT4 quantization is applied as the INT4 only has 16 numbers and a 10x smaller range leads to 2 (or 3) numbers for the representations of those smaller-range rows.

This analysis results also indicate why more expensive hidden-states knowledge distillation [2, 36] is used for ultra-low precision quantization to close the accuracy gap. However, as the training cost of knowledge distillation for large-scale models is too high, a lightweight and efficient method is desirable for PTQ.

## 4 Methodology

### 4.1 Fine-grained Hardware-friendly Quantization Scheme

As shown in Section 3, even applying INT8 PTQ to BERT/GPT-3-style models leads to significant accuracy degradation. The key challenge is the representation of INT8 cannot fully capture the different numerical ranges of different rows in weight matrices and different activation tokens. One way to address this is to use group-wise (token-wise) quantization for the weight matrix (activations).

**Group-wise Quantization for Weights** Group-wise weight matrix quantization has first been proposed in [56], where a weight matrix  $\mathbf{W} \in \mathbb{R}^{n \times m}$  is partitioned into  $g$  groups, and each group is quantized separately. However, in [56], the authors only apply this for quantization aware training. More importantly, they do not consider the hardware efficiency constraint and they do not have a system backend support. As such, they lack the real latency reduction benefit.

In our design, we consider the hardware constraint from Ampere Architecture of GPUs (e.g, A100), where the compute unit is based on Warp Matrix Multiply and Accumulate (WMMA) tiling size [53] to achieve the best speedup. Later, we will show that our group-wise quantization leads to much better accuracy as

compared to single-matrix quantization due to its finer-granularity quantization while still achieving great latency reduction.

**Token-wise Quantization for Activations** As mentioned in Section 3 and Appendix A.2, a common practice for existing PTQ work is to use static quantization for activation, where the min/max range is calculated at an offline calibration phase. Such a method might be sufficient for small scale models where the variance in the activation range is small. However, as analyzed in Section 3, there is a huge variance in the activation range for large-scale transformer models such as GPT-3<sub>350M</sub> and BERT<sub>base</sub>. As such, a static quantization scheme (often applied to all tokens/samples) would lead to significant accuracy drop. One natural idea to overcome this issue is to adopt finer-grained token-wise quantization and dynamically calculate the min/max range for each token to reduce the quantization error from activations. Our evaluation in Section 5 also shows that token-wise quantization for activation significantly improves the accuracy of GPT-3-style and BERT models.

However, directly applying token-wise quantization using existing DL frameworks, such as the PyTorch quantization suite, would lead to significant quantization and dequantization cost because token-wise quantization introduces additional operations that lead to expensive data movement overhead between the GPU compute units and the main memory. To address this issue, we build a highly optimized inference backend for token-wise quantization of transformer models. For example, the inference backend of ZeroQuant employs so called *kernel fusion* technique to fuse quantization operator with its previous operator, like layer normalization, to alleviate the data movement cost from token-wise quantization. Similarly, the dequantization cost of the different GeMMs' output is alleviated by scaling the INT32 accumulation using both the weight and activation quantization scales, before writing the final FP16 result back to the main memory for the next FP16 operator (like GeLU). Those optimization will be discussed in more details in Section 4.3.

Token-wise quantization can significantly reduce the representation error for quantized activations. Also, as it does not need to calibrate the activation range, later we will show that there is no quantization-related cost (e.g., activation range calibration) for a moderate quantization scheme (INT8 weight with INT8 activation) for ZeroQuant.

## 4.2 Layer-by-layer Knowledge Distillation with Affordable Cost

Knowledge distillation (KD) is one of the most powerful methods to alleviate the accuracy degradation after model compression. However, there are several limitations of KD, especially for hidden-states KD on large-scale language models: (1) KD needs to hold a teacher and a student model together during the training, which dramatically increases the memory and compute cost; (2) KD usually requires full training of the student model. Therefore, several copies (gradient, first/second order momentum) of the weight parameters need to be stored in memory to update the model; (3) KD generally requires original training data, which sometimes are not accessible due to privacy/confidential issues.

To address those limitations, we present our layer-by-layer distillation (LKD) algorithm. Assume the target model for quantization has  $N$  transformer blocks,  $L_1, \dots, L_N$ , the accessible dataset has input  $(\mathbf{X}, \mathbf{Y})$ , which can be the original training data or datasets from other resources. Our LKD quantizes the network layer-by-layer and uses its original (i.e., unquantized) version as the teacher model. More specifically, assume layer  $L_k$  is going to be quantized, and its quantized version is  $\hat{L}_k$ . Then we use the output of the  $L_{k-1}$  (i.e., by running inference on  $X$  over the first  $k-1$  layers) as the input of  $L_k$  and  $\hat{L}_k$ , measure the difference, and do the model update to  $L_k$ , i.e.,

$$\mathcal{L}_{LKD,k} = MSE \left( L_k \cdot L_{k-1} \cdot L_{k-2} \cdot \dots \cdot L_1(\mathbf{X}) - \hat{L}_k \cdot L_{k-1} \cdot L_{k-2} \cdot \dots \cdot L_1(\mathbf{X}) \right), \quad (1)$$

where  $MSE$  is the mean square loss, and it can be also replaced by other losses (e.g., KL divergence) as well. As can be seen, (1) our LKD does not need to hold a separate teacher as we use the same  $L_1$  to  $L_{k-1}$  for both teacher/student model. As such, the only extra model cost we have is  $L_k$ ; (2) the memory overhead of optimizer states are significantly reduced as the only optimizing layer is  $L_k$ ; (3) as we never optimize the end-to-end model, the training does not depend on the label anymore. Later, we will show that LKD does not rely on the original training data in Section 5.6.



Figure 2: The illustration of normal (left) and our fused (right) INT8 GeMM.

Table 2: Result of BERT<sub>base</sub> on the development set of GLUE benchmark (except WNLI). [56]<sup>+</sup> uses 128 groups for weight matrix which is hard to get GPU acceleration. [6]<sup>\*</sup> uses mixed INT8 and FP16 activation, and it directly reports the average metric of MNLI/MRPC/QQP/STS-B, which is basically the average of the two metrics we used for our runs.

Precision (Method)	CoLA	MNLI-m	MNLI-mm	MRPC	QNLI	QQP	RTE	SST-2	STS-B	Ave.	Ave. Time (s)
W16A16 (Baseline)	59.72	84.94	85.06	86.27/90.57	92.15	91.51/88.56	72.20	93.23	90.06/89.59	83.95	N/A
W8A8 [56] (QAT) <sup>+</sup>	—	83.91	83.83	—	—	—	—	92.83	—	—	—
W8A8 [76] (QAT)	58.48	—	—	—/89.56	90.62	—/87.96	68.78	92.24	89.04/—	—	—
W8A8 (QAT)	61.21	84.80	84.64	83.82/88.85	91.29	91.29/88.28	71.12	92.89	88.39/88.18	83.37	2900
W8A8 (PTQ)	56.06	79.99	81.06	75.49/79.67	87.35	89.92/86.82	48.38	91.40	86.58/86.44	77.41	6
W8A8/16 [6] (PTQ)*	58.63	82.67	82.67	88.74	90.41	89.40	68.95	92.66	88.00	82.46	Unknown
W8A8 (ZeroQuant)	59.59	84.83	85.13	86.03/90.39	91.98	91.45/88.46	71.12	93.12	90.09/89.62	83.75	0
W4/8A16 (PTQ)	0.00	16.74	16.95	31.62/0.00	50.74	63.18/0.00	47.29	70.64	16.48/15.91	33.11	6
W4/8A16 (ZeroQuant)	57.29	82.69	83.27	84.56/88.40	90.04	86.52/79.49	70.76	92.78	88.46/88.61	81.65	0
W4/8A16 (ZeroQuant-LKD)	58.50	83.16	83.69	84.80/89.31	90.83	88.94/84.12	70.04	92.78	88.49/88.67	82.35	31
W4/8A8 (ZeroQuant)	56.69	82.46	83.06	84.07/88.03	90.13	87.04/80.50	70.76	92.78	88.07/88.44	81.55	0
W4/8A8 (ZeroQuant-LKD)	58.80	83.09	83.65	85.78/89.90	90.76	89.16/84.85	71.84	93.00	88.16/88.55	82.71	31

### 4.3 Quantization-Optimized Transformer Kernels

Both optimizing the inference latency and model size is crucial for serving large-scale transformer models in practice. During inference, the batch size is often relatively small, so the inference latency of the model primarily depends on the time of loading inference needed data from the main memory. By quantizing the weights and activations to lower precision, we reduce the data volume needed to load those data, which allows more effective use of memory bandwidth and higher loading throughput. However, simply converting weights/activations to INT8 does not guarantee improved latency because there are additional data movement overhead associated with quantization/dequantization operations as shown in Figure 2 (red box). Such an overhead becomes expensive and in some cases surpasses the performance benefits of using low precision. To reap the accuracy improvement from token-wise quantization while obtaining improved latency, we now present our optimizations that maximize the memory bandwidth utilization to speed up inference latency for ZeroQuant.

**CUTLASS INT8 GeMM** To support INT8 computation, we use CUTLASS [5] INT8 GeMM implementation tuned for different batch sizes. Unlike standard GPU backend library, such as cuDNN, using CUTLASS allows us to more flexibly fuse quantization operation before and after GeMM to reduce kernel launching and data-movement overhead.

**Fusing Token-wise Activation Quantization** Token-wise quantization/dequantization introduce many additional operations that lead to extra data movement cost. To eliminate these cost, we use *kernel fusion* [67] to fuse quantization operation for activation with its previous element-wise and/or reduction operations such as bias-add, GeLU, and LayerNorm into a single operator, as illustrated by the green box in Figure 2. For the dequantization operation (e.g., dequantizing the integer output from the GeMM operator), we similarly fuse it with our custom GeMM schedule to avoid additional read/write accesses to the main memory as illustrated by the blue box in Figure 2.

By doing the above optimizations, we are able to show significant latency reduction for BERT and GPT-3-style models in Section 5. Please see Appendix D for more details about our system optimization.

## 5 Results

**Experimental Details** To evaluate the proposed ZeroQuant, we test it on both BERT and GPT-3 models. For BERT, we tested both BERT<sub>base</sub> and BERT<sub>large</sub> on GLUE benchmark; and for GPT-3-style models, we tested the GPT-3<sub>350M</sub> (i.e., GPT-3-style model with 350M parameters) and GPT-3<sub>1.3B</sub> (i.e., GPT-3-style model with 1.3B parameters) on 20 zero-shot evaluation tasks, including 19 accuracy-based tasks and 1 language modeling generation task. To illustrate the scalability of the proposed ZeroQuant, we also directly apply it to two of the largest open-sourced GPT-3-style models, i.e., GPT-J<sub>6B</sub> [66] and GPT-NeoX<sub>20B</sub> [4]. We use a fixed set of hyperparameters for all the LKD-related experiments even though tuning them may benefit our results. Please see Appendix B.2 for more training details and see Appendix B.3 for the reported metrics for BERT. To provide a comprehensive study, we also include a tuning result in Appendix E on BERT and an ablation study for different proposed components in Section 5.5.

**Notation Explanation** We use WxAy to represent using x-bit for weight quantization and y-bit for activation quantization. Unless specific explanation, for W4/8, we quantize the MHSA’s weight to INT8 and FFC’s weight to INT4; for A8/16, we use FP16 activation for self-attention calculation (i.e., the GeMM related to  $\mathbf{W}_{q/k/v}$ ) and use INT8 for the rest calculation. We use ZeroQuant to represent the method with only fine-grained quantization schemes and use ZeroQuant-LKD to represent the method with both fine-grained quantization schemes and LKD.

### 5.1 Main Results of BERT

**BERT<sub>base</sub>** We report the results of BERT<sub>base</sub> in Table 2. For W8A8, the average accuracy of PTQ degrades more than 10 points. However, ZeroQuant can achieve 83.75 scores, which is only 0.2 lower than baseline. Particularly, as ZeroQuant has no activation range calibration phase, the cost of ZeroQuant is 0 which is even cheaper than standard PTQ. As compared to [6], our method achieves a better average score (1.29 higher). Meanwhile, as compared to INT8 activation used in ZeroQuant, [6] uses mixed INT8 and FP16 activation.

We also compare our method with our internal trained QAT and other QAT works [56, 76]. As can be seen, with comparable accuracy results as those QAT methods, ZeroQuant can save the retraining cost from 2900s to 0s for INT8 quantization.

For the more aggressive weight quantization with minimal (or no) training quantization, i.e., W4/8A16, PTQ fully loses all accuracy (pure random prediction). However, ZeroQuant can still achieve an 81.65 average score. On top of ZeroQuant, if we add our LKD, the accuracy can be further boosted to 82.35 with a cost of 31s per task using only a single GPU, which is 93.5x cheaper than INT8 QAT quantization. We also test ZeroQuant and ZeroQuant-LKD under the W4/8A8 quantization scheme and both of them achieve similar accuracy performance as W4/8A16. If hyper-parameter tuning is applied to LKD, ZeroQuant-LKD can achieve an 83.22 average score under W4/8A8, which is similar to QAT’s W8A8 result. Please see Appendix E for more details.

**BERT<sub>large</sub>** We test our methods on BERT<sub>large</sub> as well and the results are shown in Table 3. Similar to BERT<sub>base</sub>, ZeroQuant achieves much better accuracy than PTQ methods. As compared to QAT methods, ZeroQuant has comparable results on larger datasets (like MNLI/QQP) and has better performance on small tasks (e.e., CoLA/MRPC/RTE). We actually tune QAT for multiple learning rates but cannot get even better performance for those small tasks (see Appendix F for more details).

For more aggressive quantization schemes, like W4/8A16 and W4/8A8, ZeroQuant and ZeroQuant-LKD still achieve good accuracy except for RTE but the model size is about 3x smaller than FP16 counterpart. This is aligned with the INT8 QAT results, which lose significantly more accuracy on RTE. Thanks to the lightweight cost of LKD, it only takes about 550s to finish each task even on BERT<sub>large</sub>, which is 13x cheaper than QAT.

### 5.2 Main Results of GPT-3-style Models

**GPT-3<sub>350M</sub>** We first test ZeroQuant and ZeroQuant-LKD on GPT-3<sub>350M</sub> and report the result in Table 4. The first interesting finding of zero-shot evaluation on GPT-3-stype models is that the accuracy performance

Table 3: Result of BERT<sub>large</sub> on the development set of GLUE benchmark (except WNLI). <sup>+</sup>We extensively tuned the learning rate for QAT (see Appendix F for more details).

Precision (Method)	CoLA	MNLI-m	MNLI-mm	MRPC	QNLI	QQP	RTE	SST-2	STS-B	Ave.	Ave. Time (s)
W16A16 (Baseline)	63.35	86.65	85.91	87.99/91.62	92.24	91.08/88.08	74.01	93.46	90.34/90.11	85.03	N/A
W8A8 [76] (QAT)	—	—	—	—/90.9	91.74	—	—	—	90.12/—	—	—
W8A8 (QAT) <sup>+</sup>	59.85	86.65	86.35	85.29/89.43	92.55	91.60/88.60	61.37	93.23	87.55/87.65	82.78	7181
W8A8 (PTQ)	60.57	75.69	76.94	81.13/84.93	88.49	84.04/74.35	46.93	91.74	62.75/55.77	73.54	31
W8A8 (ZeroQuant)	63.38	86.52	85.64	87.75/91.50	92.31	91.09/88.05	72.56	93.35	90.45/90.19	84.81	0
W4/8A16 (PTQ)	0.00	16.85	33.24	68.38/80.89	51.25	63.18/0.00	52.71	52.41	-5.74/-8.51	35.73	31
W4/8A16 (ZeroQuant)	62.99	84.77	84.42	87.50/91.16	91.63	90.03/86.41	48.01	92.16	89.49/89.28	81.23	0
W4/8A16 (ZeroQuant-LKD)	63.72	84.90	84.81	87.99/91.39	91.45	90.34/86.92	51.62	92.43	89.46/89.29	81.85	550
W4/8A8 (ZeroQuant)	62.34	84.62	84.25	87.75/91.38	91.87	89.86/86.09	47.65	91.97	89.39/89.17	81.06	0
W4/8A8 (ZeroQuant-LKD)	63.51	84.70	84.71	88.73/91.99	91.73	90.25/86.74	49.82	92.09	89.34/89.08	81.62	550

Table 4: Post training quantization result of GPT-3<sub>350M</sub> on 20 zero-shot evaluation datasets. Please see Table H.1 for the results of all 20 tasks.

Precision (Method)	Lambada ( $\uparrow$ )	PIQA ( $\uparrow$ )	OpenBookQA ( $\uparrow$ )	RTE ( $\uparrow$ )	ReCoRd ( $\uparrow$ )	Ave. 19 Tasks ( $\uparrow$ )	Wikitext-2 ( $\downarrow$ )	Time Cost
W16A16	49.3	66.3	29.4	53.8	75.1	38.9	21.5	N/A
W8A8 (PTQ)	42.6	64.1	28.0	53.1	67.5	37.8	26.2	7 mins
W8A8 (ZeroQuant)	51.0	66.5	29.2	53.4	74.9	38.7	21.7	0
W4/8A16 (PTQ)	0.00	51.4	30.2	52.7	16.1	28.9	1.76e5	7 mins
W4/8A16 (ZeroQuant)	10.1	58.5	27.2	52.0	56.5	33.5	88.6	0
W4/8A16 (ZeroQuant-LKD)	39.8	63.8	29.4	53.1	70.1	37.0	30.6	1.1 hours
W4/8A8 (ZeroQuant)	10.5	57.7	28.0	52.7	55.3	33.4	92.1	0
W4/8A8 (ZeroQuant-LKD)	37.4	61.8	28.2	53.1	68.5	36.6	31.1	1.1 hours

of accuracy-based tasks is more tolerant to quantization than generation tasks. For instance, W8A8 PTQ has a 1.1% average accuracy drop on 19 accuracy-based tasks as compared to 4.7 points loss on Wikitext-2. Comparing ZeroQuant with PTQ using W8A8, we can reduce the accuracy gap from 1.1% to 0.2% and the perplexity (PPL) gap from 4.7 to 0.2 with no activation range calibration cost.

For W4/8A16 quantization scheme, PTQ can hardly predict reasonable answers for the majority of tasks and its generation performance on Wikitext-2 is fully crashed. As a comparison, ZeroQuant still achieves non-trivial performance on some tasks but its generation performance significantly degrades on Wikitext-2. LKD brings a significant performance boost for this W4/8A16 setting. Note that ZeroQuant-LKD increases the accuracy from 33.5 to 37.0 and decreases the PPL from 88.6 to 30.6 compared to ZeroQuant, and the entire cost of this is just 3.1 hours on a single A100 GPU. Note that this is about 0.027% GPU hours of the full pretraining cost (128 A100 GPUs for 32 hours). Similar to W4/8A16, ZeroQuant-LKD achieves much better performance than ZeroQuant on W4/8A8 by using the lightweight LKD.

**GPT-3<sub>1.3B</sub>** The results of GPT-3<sub>1.3B</sub> are shown in Table 5. Similar to GPT-3<sub>350M</sub>, for W8A8, ZeroQuant has much better performance than PTQ with less no activation calibration cost, particularly for the generation task Wikitext-2 (3.2 points lower). Also, for W4/8 quantization, LKD can bring non-trivial performance gain for ZeroQuant. The cost of LKD is about 0.02% of the full pre-training cost (128 A100 GPUs for 120 hours)

### 5.3 Latency Reduction of BERT and GPT-3-style Models

We compare the inference speed of BERT between FP16 and our INT8 versions in Table 6 on a single 40G-A100 GPU. Using our efficient quantization kernel implementation and operator fusion, the INT8 model can achieve 2.27–5.19x speedup on BERT<sub>base</sub> and 2.47–5.01x on BERT<sub>large</sub>.

We also include the latency comparison of GPT-3-style models between FP16 and our INT8 version. Particularly, we use the model to generate the first 50 tokens based on a given text and measure the average latency. Our INT8 model leads to 4.16x/4.06x speedup for GPT-3<sub>350M</sub>/GPT-3<sub>1.3B</sub> as compared to the FP16 counterpart.

Table 5: Post training quantization result of GPT-3<sub>1.3B</sub> on 20 zero-shot evaluation datasets. Please see Table H.2 for the results of all 20 tasks.

Precision (Method)	Lambada ( $\uparrow$ )	PIQA ( $\uparrow$ )	OpenBookQA ( $\uparrow$ )	RTE ( $\uparrow$ )	ReCoRd ( $\uparrow$ )	Ave. 19 Tasks ( $\uparrow$ )	Wikitext-2 ( $\downarrow$ )	Time Cost
W16A16	61.3	71.4	33.6	53.1	82.6	42.4	15.3	N/A
W8A8 (PTQ)	54.8	67.7	16.6	54.5	75.7	40.5	18.9	13 mins
W8A8 (ZeroQuant)	62.6	70.7	33.4	52.7	80.9	42.3	15.7	0
W4/8A16 (PTQ)	0.00	50.4	27.0	50.9	15.8	29.0	1.35e5	13 mins
W4/8A16 (ZeroQuant)	43.9	66.5	30.0	52.7	77.3	39.38	21.9	0
W4/8A16 (ZeroQuant-LKD)	59.4	69.5	31.6	52.7	79.7	41.5	17.6	3 hours
W4/8A8 (ZeroQuant)	46.8	66.4	28.8	52.7	76.2	39.24	24.1	0
W4/8A8 (ZeroQuant-LKD)	48.7	68.1	29.0	52.0	77.4	39.90	18.2	3 hours

Table 6: The speedup of our W8A8 as compared to W16A16. We measure the end-to-end average latency for the entire BERT model, and the time reported is in milliseconds.

Seq Len BS	Precision	128							256								
		1	2	4	8	16	16	64	128	1	2	4	8	16	16	64	128
BERT <sub>base</sub>	W16A16	2.45	3.22	3.85	5.51	9.96	17.93	34.25	67.08	3.13	4.05	5.70	10.55	19.27	36.69	71.75	140.0
	W8A8	1.08	1.16	1.42	1.76	2.58	3.90	6.74	12.92	1.22	1.44	2.08	2.88	4.10	7.80	14.66	28.13
	Speedup	2.27	2.78	2.71	3.13	3.86	4.60	5.08	5.19	2.57	2.81	2.74	3.66	4.70	4.70	4.89	4.98
BERT <sub>large</sub>	W16A16	5.45	6.38	8.73	13.88	26.34	48.59	92.49	183.4	6.39	8.94	14.66	27.99	51.94	98.78	195.9	384.5
	W8A8	2.08	2.58	2.84	3.79	6.21	10.28	18.86	36.62	2.55	3.36	4.16	6.88	11.61	21.20	41.24	79.90
	Speedup	2.62	2.47	3.07	3.66	4.24	4.73	4.90	5.01	2.51	2.66	3.52	4.07	4.47	4.66	4.75	4.81

## 5.4 A Showcase of GPT-J<sub>6B</sub> and GPT-NeoX<sub>20B</sub>

To demonstrate the scalability of ZeroQuant, we applied it to two of the largest open-sourced models, i.e., GPT-J<sub>6B</sub> and GPT-NeoX<sub>20B</sub>, which have 6B and 20B parameters separately.

We report the results of GPT-J<sub>6B</sub> in Table 7 on three generation datasets, i.e., PTB [41], Wikitext-2, and Wikitext-103 [42]. As can be seen, as compared to FP16 precision, ZeroQuant achieves similar PPL on all three different tasks. To compare the latency, we again use the average latency number to generate the first 50 tokens. Our W8A8 can get up to 3.67x speedup compared to the FP16 version.

To quantize GPT-NeoX<sub>20B</sub> to W8A8 for all GeMMs, the accuracy significantly decreases. We retrieve the quantization of each weight matrix and of each activation, and finally find out that the activation quantization for the attention calculation (i.e., the input of self-attention) causes the accuracy loss. We conjecture that this is because of the sensitivity of the self-attention module for extra-large models (20B) but cannot verify this for other models due to the lack of open-sourced extra-large models and the full evaluation pipeline. As such, we leave the input activation for self-attention in FP16 and quantize the rest to INT8. The results are shown in Table 8. Our W8A8/16 achieves similar accuracy performance but can reduce both the GPU resource requirement (from 2 A100 GPUs to 1) and the latency from 65ms to 25ms, which together lead to 5.2x better throughput/efficiency.

## 5.5 Ablation Study of Different Components

To investigate the performance gain of each component we introduced in Section 4, i.e., group-wise weight quantization, token-wise activation quantization, and lightweight layer-by-layer knowledge distillation, we here do an ablation study on BERT<sub>large</sub> with W4/8A8.

We present the results in Table 9. As can be seen, group-wise weight quantization boosts the accuracy (random-guess prediction) from PTQ to a non-trivial result (66.52). Further adding token-wise quantization improves 14.54 points accuracy performance. On top of those (i.e., ZeroQuant), LKD further brings a 0.56 point gain.

Table 7: Post training quantization result of GPT-J<sub>6B</sub> on three zero-shot generation tasks

Precision	PTB	Wikitext-2	Wikitext-103	Latency
W16A16	20.47	10.35	10.35	29.13ms (1x)
W8A8	20.97	10.51	10.52	7.94ms (3.67x)

Table 8: Post training quantization result of GPT-NeoX<sub>20B</sub> on 19 zero-shot evaluation datasets. Please see Table H.4 for the results of all 19 tasks.

Precision	Lambada	PIQA	Ave.	19 Tasks	Latency
W16A16	71.7	77.7	50.5	2×65ms (1x)	
W8A8/16	71.9	78.3	50.4	1×25ms (5.2x)	

Table 9: Ablation study of different components for BERT<sub>large</sub> on the development set of GLUE. The quantization scheme used here is W4/8A8. Here, GP is the abbreviation of group-wise weight quantization, TQ is the abbreviation of token-wise activation quantization.

GQ	TQ	LKD	CoLA	MNLI-m	MNLI-mm	MRPC	QNLI	QQP	RTE	SST-2	STS-B	Ave.
✗	✗	✗	-0.79	33.07	32.94	68.38/80.54	49.42	63.18/0.00	52.71	52.29	-4.27/-1.90	35.85
✓	✗	✗	59.81	66.63	68.79	68.63/71.17	83.87	78.24/61.30	46.93	89.45	54.58/32.52	66.52
✓	✓	✗	62.34	84.62	84.25	87.75/91.38	91.87	89.86/86.09	47.65	91.97	89.39/89.17	81.06
✓	✓	✓	63.51	84.70	84.71	88.73/91.99	91.73	90.25/86.74	49.82	92.09	89.34/89.08	81.62

Table 10: Post training quantization result of GPT-3<sub>350M</sub> on 20 zero-shot evaluation datesets The quantization scheme here is W4/8A8. Please see Table H.3 for the results of all 20 tasks.

Method	Data Resource	Lambada (↑)	PIQA (↑)	OpenBookQA (↑)	RTE (↑)	ReCoRd (↑)	Ave. 19 Tasks (↑)	Wikitext-2 (↓)
ZeroQuant	—	10.5	57.7	28.0	52.7	55.3	33.4	92.1
ZeroQuant-LKD	Random data	26.1	59.3	29.2	50.5	64.9	34.5	40.6
ZeroQuant-LKD	Wikipedia	33.9	62.4	28.0	52.7	69.5	36.2	30.4
ZeroQuant-LKD	Original data	37.4	61.8	28.2	53.1	68.5	36.6	31.1

## 5.6 No Access to The Original Training Data

As mentioned in previous sections, the original training data are oftentimes hard to access due to the privacy and/or confidential issues. Therefore, we here study the performance of our LKD when there is no direct access to the original training data. As the distillation objective of our LKD does not depend on the label, the training data used for LKD can be very flexible.

We compare the performance of GPT-3<sub>350M</sub> on W4/8A8 quantization scheme using three different training data resources, i.e., random data (using random integer number to generate token ids), Wikipedia (using Huggingface to get the data<sup>1</sup>), and original PILE dataset.

The results are shown in Table 10. Compared to ZeroQuant, LKD using random data can boost the accuracy by 1.1% and reduce the PPL from 92.1 to 40.6. The reason why random data can still significantly improve the performance is that LKD does not optimize the end-to-end pipeline and it only layer-by-layer learns the internal dependency from the teacher model. Therefore, random data can also provide meaningful information. Using Wikipedia data from Huggingface can further improve the accuracy to 36.2 and reduce the PPL to 30.4, which is comparable to the results using the original data. This indicates that a clean text dataset can be used for LKD when we do not have access to the original full dataset.

## 6 Conclusions

With the rapid growth of large model sizes, we have reach a point to consider how to serve those models in practice. Although several works demonstrate that post-training quantization can be applied to BERT models, to the best of our knowledge, there have been no existing works on (1) billion-scale GPT-3-style models, (2) ultra-low precision post-training quantization, and (3) end-to-end solution of how to efficiently

<sup>1</sup><https://huggingface.co/datasets/wikipedia>

serve the quantized model online. In this work, we offer fine-grained compression schemes for both weight and activations to enable INT8 quantization for up to 20B-scale models (GPT-NeoX<sub>20B</sub>). We also offer a novel affordable layer-by-layer knowledge distillation for ultra-low precision quantization, which leads to 3x model size reduction compared to FP16 model while achieving minimal accuracy degradation. Furthermore, we provide a system backend support and show up to 5.19x speedup on BERT models and 5.2x better efficiency on GPT-NeoX<sub>20B</sub>.

## Acknowledgments

This work is done within the DeepSpeed team in Microsoft. We appreciate the help from the DeepSpeed team. Particularly, we thank Jeff Rasley and Elton Zheng for solving the engineering issue. We thank the engineering supports from the Turing team in Microsoft.

## References

- [1] Ardavan Afshar, Ioakeim Perros, Evangelos E Papalexakis, Elizabeth Searles, Joyce Ho, and Jimeng Sun. Copac: Constrained parafac2 for sparse & large datasets. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 793–802, 2018.
- [2] Haoli Bai, Wei Zhang, Lu Hou, Lifeng Shang, Jing Jin, Xin Jiang, Qun Liu, Michael Lyu, and Irwin King. Binarybert: Pushing the limit of bert quantization. *arXiv preprint arXiv:2012.15701*, 2020.
- [3] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, Seattle, Washington, USA, October 2013. Association for Computational Linguistics.
- [4] Sid Black, Stella Biderman, Alex Andonian, Quentin Anthony, Preetham Gali, Leo Gao, Eric Hallahan, Josh Levy-Kramer, Connor Leahy, Lucas Nestler, Kip Parker, Jason Phang, Michael Pieler, Shivanush Purohit, Tri Songz, Phil Wang, and Samuel Weinbach. GPT-NeoX: Large scale autoregressive language modeling in pytorch, 2021.
- [5] NVIDIA blog. CUTLASS: Fast Linear Algebra in CUDA C++. <https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/>, December 2017.
- [6] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Understanding and overcoming the challenges of efficient transformer quantization. *arXiv preprint arXiv:2109.12948*, 2021.
- [7] Michael Boratko, Harshit Padigela, Divyendra Mikkilineni, Pritish Yuvraj, Rajarshi Das, Andrew McCallum, Maria Chang, Achille Fokoue-Nkoutche, Pavan Kapanipathi, Nicholas Mattei, et al. A systematic classification of knowledge, reasoning, and context within the arc dataset. *arXiv preprint arXiv:1806.00358*, 2018.
- [8] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.
- [9] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Zeroq: A novel zero shot quantization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13169–13178, 2020.
- [10] Daniel Cer, Mona Diab, Eneko Agirre, Inigo Lopez-Gazpio, and Lucia Specia. Semeval-2017 task 1: Semantic textual similarity-multilingual and cross-lingual focused evaluation. *arXiv preprint arXiv:1708.00055*, 2017.
- [11] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.
- [12] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*, 2019.
- [13] Ido Dagan, Dan Roth, Mark Sammons, and Fabio Massimo Zanzotto. Recognizing textual entailment: Models and applications. *Synthesis Lectures on Human Language Technologies*, 6(4):1–220, 2013.
- [14] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

- [15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [16] Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, and Noah Smith. Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping. *arXiv preprint arXiv:2002.06305*, 2020.
- [17] William B Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*, 2005.
- [18] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. HAWQ: Hessian aware quantization of neural networks with mixed-precision. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 293–302, 2019.
- [19] Steven K Esser, Jeffrey L McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S Modha. Learned step size quantization. *arXiv preprint arXiv:1902.08153*, 2019.
- [20] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.
- [21] Angela Fan, Pierre Stock, Benjamin Graham, Edouard Grave, Remi Gribonval, Herve Jegou, and Armand Joulin. Training with quantization noise for extreme fixed-point compression. *arXiv preprint arXiv:2004.07320*, 2020.
- [22] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.
- [23] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.
- [24] Mitchell A Gordon, Kevin Duh, and Nicholas Andrews. Compressing bert: Studying the effects of weight pruning on transfer learning. *arXiv preprint arXiv:2002.08307*, 2020.
- [25] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.
- [26] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *Workshop paper in NIPS*, 2014.
- [27] Shankar Iyer, Nikhil Dandekar, and Kornl Csernai. First quora dataset release: Question pairs.(2017). URL <https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs>, 2017.
- [28] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351*, 2019.
- [29] Jing Jin, Cai Liang, Tiancheng Wu, Liqin Zou, and Zhiliang Gan. Kdlsq-bert: A quantized bert combining knowledge distillation with learned step size quantization. *arXiv preprint arXiv:2101.05938*, 2021.
- [30] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*, 2017.
- [31] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. I-bert: Integer-only bert quantization. In *International conference on machine learning*, pages 5506–5518. PMLR, 2021.
- [32] Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. Race: Large-scale reading comprehension dataset from examinations. *arXiv preprint arXiv:1704.04683*, 2017.

- [33] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.
- [34] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.
- [35] Hector Levesque, Ernest Davis, and Leora Morgenstern. The winograd schema challenge. In *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*. Citeseer, 2012.
- [36] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.
- [37] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.
- [38] Zhenhua Liu, Yunhe Wang, Kai Han, Wei Zhang, Siwei Ma, and Wen Gao. Post-training quantization for vision transformer. *Advances in Neural Information Processing Systems*, 34, 2021.
- [39] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *Workshop paper in CVPR*, 2017.
- [40] Yihuan Mao, Yujing Wang, Chufan Wu, Chen Zhang, Yang Wang, Yaming Yang, Quanlu Zhang, Yunhai Tong, and Jing Bai. Ladabert: Lightweight adaptation of bert through hybrid model compression. *arXiv preprint arXiv:2004.04124*, 2020.
- [41] Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Using Large Corpora*, page 273, 1994.
- [42] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017.
- [43] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *arXiv preprint arXiv:1905.10650*, 2019.
- [44] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. *arXiv preprint arXiv:1809.02789*, 2018.
- [45] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or down? adaptive rounding for post-training quantization. In *International Conference on Machine Learning*, pages 7197–7206. PMLR, 2020.
- [46] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1325–1334, 2019.
- [47] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*, 2016.
- [48] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.
- [49] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019.
- [50] Alessandro Raganato, Yves Scherrer, and Jörg Tiedemann. Fixed encoder self-attention patterns in transformer-based machine translation. *arXiv preprint arXiv:2002.10260*, 2020.

- [51] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.
- [52] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.
- [53] Greg Ruetsch. Using tensor cores in cuda fortran. *Nvidia Blog*, 2021.
- [54] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8732–8740, 2020.
- [55] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.
- [56] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-BERT: Hessian based ultra low precision quantization of bert. In *AAAI*, pages 8815–8821, 2020.
- [57] Shaden Smith, Mostafa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.
- [58] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.
- [59] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. Patient knowledge distillation for bert model compression. *arXiv preprint arXiv:1908.09355*, 2019.
- [60] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. *arXiv preprint arXiv:2004.02984*, 2020.
- [61] Chaofan Tao, Lu Hou, Wei Zhang, Lifeng Shang, Xin Jiang, Qun Liu, Ping Luo, and Ngai Wong. Compression of generative pre-trained language models via quantization. *arXiv preprint arXiv:2203.10705*, 2022.
- [62] Sandeep Tata and Jignesh M Patel. Piqa: An algebra for querying protein data sets. In *15th International Conference on Scientific and Statistical Database Management, 2003.*, pages 141–150. IEEE, 2003.
- [63] Ian Tenney, Dipanjan Das, and Ellie Pavlick. Bert rediscovers the classical nlp pipeline. *arXiv:1905.05950*, 2019.
- [64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
- [65] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.
- [66] Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. <https://github.com/kingoflolz/mesh-transformer-jax>, May 2021.

- [67] Guibin Wang, Yisong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In Peidong Zhu, Lizhe Wang, Feng Xia, Huajun Chen, Ian McLoughlin, Shiao-Li Tsao, Mitsuhsisa Sato, Sun-Ki Chai, and Irwin King, editors, *2010 IEEE/ACM Int'l Conference on Green Computing and Communications, GreenCom 2010, & Int'l Conference on Cyber, Physical and Social Computing, CPSCom 2010, Hangzhou, China, December 18-20, 2010*, pages 344–350. IEEE Computer Society, 2010.
- [68] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *arXiv preprint arXiv:2002.10957*, 2020.
- [69] Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. Neural network acceptability judgments. *arXiv preprint arXiv:1805.12471*, 2018.
- [70] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122, 2018.
- [71] Adina Williams, Tristan Thrush, and Douwe Kiela. Anlizing the adversarial natural language inference dataset. *arXiv preprint arXiv:2010.12729*, 2020.
- [72] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierrick Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. HuggingFace’s Transformers: State-of-the-art natural language processing. *ArXiv*, pages arXiv–1910, 2019.
- [73] Vikas Yadav, Steven Bethard, and Mihai Surdeanu. Quick and (not so) dirty: Unsupervised selection of justification sentences for multi-hop question answering. *arXiv preprint arXiv:1911.07176*, 2019.
- [74] Zhewei Yao, Linjian Ma, Sheng Shen, Kurt Keutzer, and Michael W Mahoney. Mlpruning: A multilevel structured pruning framework for transformer-based models. *arXiv preprint arXiv:2105.14636*, 2021.
- [75] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 811–824. IEEE, 2020.
- [76] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8BERT: Quantized 8bit bert. *arXiv preprint arXiv:1910.06188*, 2019.
- [77] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.
- [78] Sheng Zhang, Xiaodong Liu, Jingjing Liu, Jianfeng Gao, Kevin Duh, and Benjamin Van Durme. Record: Bridging the gap between human and machine commonsense reading comprehension. *arXiv preprint arXiv:1810.12885*, 2018.
- [79] Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. Ternarybert: Distillation-aware ultra-low bit bert. *arXiv preprint arXiv:2009.12812*, 2020.

## A Background

### A.1 Transformer Architecture

The transformer architecture usually has three components: an embedding layer, a stack of encoder/decoder layers, and a final classifier. In this paper, we focus on quantizing the encoder/decoder layers, i.e., the transformer block, because it is often the most memory and compute intensive components in the entire architecture. With a transformer block, there are two sub-layers, the multi-head self-attention (MHSAs) and the feed-forward connection (FFCs). We give a short review later and please refer to [64] for more details. At high level, transformer models can be broadly categorized to three branches: encoder-only models (BERT) [63], decoder-only models (GPT-3-style) [48], and encoder-decoder models (T5) [49]. In this paper, we focus on encoder-only and decoder-only models but our approach can be applied to encoder-decoder models as well.

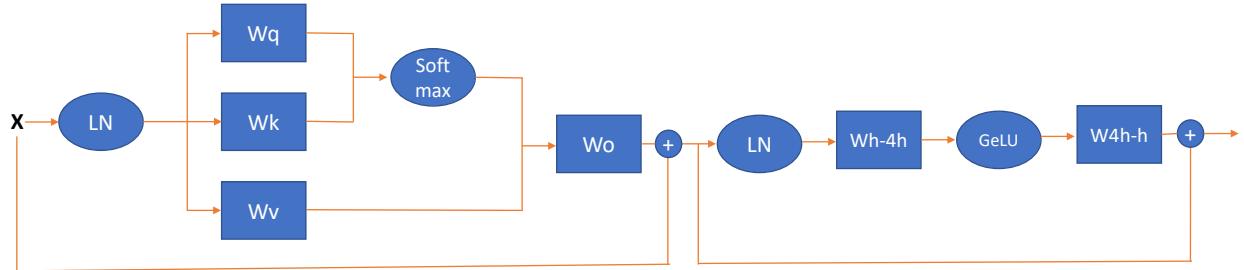


Figure A.1: The illustration of a Transformer-block.

**Transformer Block** Assume the input of an encoder layer is  $\mathbf{X}$ , the query, key, value, attention output, FFC dense, and FFC output matrices are  $\mathbf{W}_q$ ,  $\mathbf{W}_k$ ,  $\mathbf{W}_v$ ,  $\mathbf{W}_o$ ,  $\mathbf{W}_{h-4h}$ , and  $\mathbf{W}_{4h-h}$ , respectively. Then the forward propagation of a transformer-block is illustrated in Figure A.1, where LN is the layer normalization, Softmax is the softmax operator, and GeLU is the activation function.

### A.2 Quantization Background

Quantization maps high-precision numbers, e.g., FP16/FP32, to its low-precision counterpart, e.g., INT4/INT8, to reduce the model footprint and improve the compute performance. In this work, we use uniform symmetric scalar quantizers. That is to say, if we have a vector/matrix,  $\mathbf{x}$ , the quantization is applied as

$$\mathbf{x}_{\text{quantize}} = \text{round} \left( \text{clamp} \left( \frac{\mathbf{x}}{S}, -2^{bit-1}, 2^{bit-1} - 1 \right) \right), \quad (2)$$

where  $bit$  is the number of bit we use to represent the quantized value, and  $S$  is the scaling factor. For weight matrix quantization,  $S$  is generally computed as  $S = \max(\text{abs}(\mathbf{x}))$ , since the weight matrix is static during inference. On the other hand, activations' range is dynamic during inference so that an accurate  $S$  requires dynamic calculation during inference. However, to achieve best latency reduction, coarse-grained static quantization is usually applied in practice, where  $S$  is calibrated using training data (e.g., momentum based averaging) and fixed during inference [23]. Although static quantization achieves better latency reduction, it also limits the quantization representation for activations, which is discussed in Section 3.

## B Experimental Details

### B.1 Details of PTQ on BERT and GPT

For BERT, we use a batch size of 32 and sequence length 128 to calibrate the range of activations. In order to capture the dynamic range, we use 0.95 momentum with 100 iterations, i.e.,

$$x_{max} = 0.95x_{max} + 0.05\max(x_{current-iteration}), \\ x_{min} = 0.95x_{min} + 0.05\min(x_{current-iteration}).$$

For GPT-3-style models, we use the same momentum method but change the batch size to 8 with sequence length 2048.

### B.2 Details of Main Result

**BERT** BERT models are trained using the code-base from Huggingface [72]. We show our ZeroQuant method on BERT<sub>base</sub> and BERT<sub>large</sub>. We use the same lower-case tokenizer in BERT<sub>large</sub> instead of the cased tokenizer in the original paper [15]. When fine-tuning on GLUE [65] tasks ((i.e., MRPC [17], STS-B [10], SST-2 [58], QNLI [51], QQP [27], MNLI [70], CoLA [69], RTE [13]).<sup>2</sup>), we follow the instruction from Huggingface Transformer Library [72].

For ZeroQuant and ZeroQuant-LKD, we use 48 groups for group-wise weight quantization on BERT<sub>base</sub> and 64 groups for group-wise weight quantization on BERT<sub>large</sub>, for all the weight matrices.

For LKD, we use 100 iterations with batch size 32 and sequence length 128 for BERT<sub>base</sub>, and we use 400 iterations for BERT<sub>large</sub>. We fix the learning rate as 5e-6 for both models on all tasks. However, tuning them may favor ZeroQuant.

All the models are trained using a single 40G-A100 GPU (Azure ND A100 instances).

**GPT-3-style Models** All GPT-3-style models used in the paper are trained using DeepSpeed [52] and Megatron-DeepSpeed Library <sup>3</sup>. The pretraining data are from PILE dataset [22], and the training pipeline and hyperparameters are based on the Megatron-DeepSpeed repository. We use 128 A100 GPUs (Azure ND A100 instances) to do the pretraining. It takes about 32 hours to finish the training of GPT-3<sub>350M</sub> and 120 hours of GPT-3<sub>1.3B</sub>. We evaluate our results on 20 zero-shot evaluation tasks, including 19 accuracy evaluation tasks (i.e., HellaSwag [77], LAMBADA [47], TriviaQA [30], WebQS [3], Winogrande [54], PIQA [62], ARC (Challenge/Easy) [7], ANLI (R1/R2/R3) [71], OpenBookQA [44], RACE-h [32], BoolQ [12], Copa [1], RTE [13], WSC [35], MultiRC [73], ReCoRD [78]) and 1 language modeling generation task (i.e., WikiText-2 [42]).

For ZeroQuant and ZeroQuant-LKD, we use 64/128 groups for group-wise weight quantization on GPT-3<sub>350M</sub>/GPT-3<sub>1.3B</sub> for all the weight matrices.

For LKD, we use 1600 iterations with batch size 8 and sequence length 2048 for both GPT-3<sub>350M</sub> and GPT-3<sub>1.3B</sub>. We fix the learning rate as 5e-6 for both models. However, tuning them may favor ZeroQuant.

All the quantized models are trained using a single 40G-A100 GPU (Azure ND A100 instances).

### B.3 Accuracy reported for BERT on GLUE

We report the performance metric for BERT on GLUE based on Table B.1. For the average score, if the task only has one metric, we use it for the final result; if the task has two metrics, we compute the average of the two metrics first and use it for the final average score. For instance, the score of MRPC used to compute the final average is the mean of its accuracy and F1 score.

<sup>2</sup>We exclude WNLI [35] since its results are not stable [16].

<sup>3</sup><https://github.com/microsoft/Megatron-DeepSpeed>

Table B.1: Metric used for  $\text{BERT}_{\text{base}}$  on the development set of GLUE benchmark (except WNLI).

CoLA	MNLI-m	MNLI-mm	MRPC	QNLI	QQP	RTE	SST-2	STS-B
Matthews Correction	Accuracy	Accuracy / F1	Accuracy	Accuracy	Accuracy / F1	Accuracy	Accuracy	Pearson / Spearmanr

Table C.1: Post training quantization results of  $\text{BERT}_{\text{base}}$  on development sets of the GLUE benchmark (except WNLI). Here  $\text{WxAy}$  means  $x$ -bit for weight quantization and  $y$ -bit for activation quantization. Particularly, for W4/8, we quantize the MHSAs weight to INT8 and FFC’s weight to INT4. Please see Appendix B.3 for the reported metrics.

Precision	CoLA	MNLI-m	MNLI-mm	MRPC	QNLI	QQP	RTE	SST-2	STS-B	Ave.
W16A16	59.72	84.94	85.06	86.27/90.57	92.15	91.51/88.56	72.20	93.23	90.06/89.59	83.95
W8A16	60.77	84.65	84.92	85.29/89.86	91.84	91.52/88.56	71.84	93.46	89.89/89.50	83.87
W16A8	56.85	80.55	81.48	84.07/89.33	91.34	91.30/88.07	68.59	93.46	88.74/88.74	81.93
W8A8	58.74	79.99	81.06	84.31/89.51	91.18	91.24/88.03	70.76	92.66	88.33/88.73	82.16
W4/8A16	0.00	16.74	16.95	31.62/0.00	50.74	63.18/0.00	47.29	70.64	16.48/15.91	33.11

## C PTQ challenge of $\text{BERT}_{\text{base}}$

From Table C.1, we observe similar results as [6], where the accuracy degradation of INT8 quantization is mainly from activation quantization. Specifically, there is a negligible accuracy drop from INT8 weight quantization (i.e., the row of W8A16). However, with sole INT8 activation (i.e., the row of W16A8), the accuracy decreases from 84.06 to 79.61. Besides, we also push the weight quantization to a mixed-precision setting with INT4 for weights in FFC and INT8 for weights in MHSAs (i.e., the row of W4/8A16). This ultra-low precision quantization leads the model to be purely random without meaning prediction.

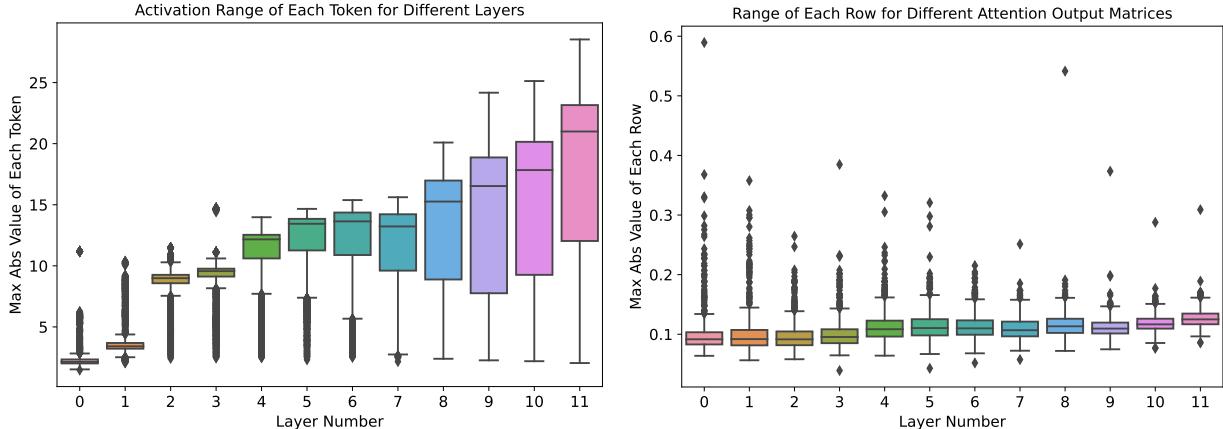


Figure C.1: The activation range of different layers (left) and the row-wise weight range of the attention output matrix ( $\mathbf{W}_o$ ) of different layers (right). The results are based on the  $\text{BERT}_{\text{base}}$  trained on MNLI dataset. Please see Figure 1 for the results of GPT-3<sub>350M</sub>.

## D Details about System Optimization

By having the weight and activation quantization, we can use the GeMM schedule that exploits the INT8 Tensor-core units which provide 2x/4x more compute efficiency compared to the FP16/FP32 Tensor cores. For this purpose, we adapt the CUTLASS library to produce multiple schedules based on the input sizes we are considering in our application, such as the batch size, sequence length, and the Transformer hidden dimension.

To achieve the best latency, we also develop our own efficient parallel implementation of the quantization operator on GPU. During the inference run-time, based on the total batch size ( $batch \times seqlen$ ), we choose the schedule that results in the lowest possible padding when performing the Tensor-core matrix-multiplication operations.

To find the best schedule for the GeMM operation, we use the CUTLASS profiler tool that explores the tiling dimensions on the thread-blocks, WARP, and WMMA (Tensor cores), as the three compute hierarchies available within the Ampere GPU architecture. Then, we find the best schedule by sorting the tile-based schedule based on either peak throughput achieved on the large-batch case, or the maximum memory bandwidth taken from the main memory when the batch size is small.

However, there are still several challenges we need to address which are discussed below.

**Operation Fusion for Token-wise Activation Quantization.** One of the main challenges of our quantization scheme is how to efficiently quantize hidden states before the GeMM operation. In order to remove the overhead, we fuse the activation quantization with its associated element-wise and/or reduction-based operations such as bias-addition, GELY, and LayerNorm. This is due to the fact that each SM takes care of one row (token) of the activation and therefore, we can reuse the computation from the thread registers and compute the quantization scale, avoiding the data movement between GPU kernels and main memory. Moreover, by converting data from FP16 to INT8, we can utilize the memory bandwidth twice, further improving the inference latency and throughput.

**Dequantization Associated with GeMM Schedule** To utilize the output of integer output from GeMM operator in the following operators, one important step is to dequantize the output by using the scaling factor of the weight and activations. This dequantization step generally introduces extra overhead for quantized network inference due to the data movement. As such, we add a custom epilogue, which converts the final accumulated result (from INT32 format) of each row and column of the output to the real value (in FP16 format), using corresponding floating-point quantization scales computed from weight and activation group-wise quantization. By fusing the dequantization with GeMM schedule, we ensure that there is no overhead exposed by using the INT8 operations while producing the FP16 results that are used in the following operation.

Furthermore, to effectively combine dequantization with the GeMM operation, we read the two groups of quantization scales for the activation and weight matrices in advance prior to completion of the multiplication of the output matrix. Doing so, we overlap the reading of the extra quantization parameters with the GeMM computation and the GeMM-plus-dequantization can seamlessly work together without stalling the inference pipeline.

**Cuda Graph Enhanced Small Model Inference.** As the execution time for specific kernels reduce by optimizing the throughput using the INT8 inference pipeline, the overhead of launching the GPU kernels and the CPU-to-GPU communication become a major bottleneck mostly on small-scale models. To address this issue, we add the CUDA-Graph support to our inference pipeline that reduces the CPU overhead, by storing the trace of the kernels launched during the inference forward computation, and creating the computation-graph to be reused in the next call to the inference pipeline. Thus, after storing the graph for the first time, we can replay the graph for the following requests, which substantially improves the performance especially on small models, such as BERT<sub>base</sub>. For a fair comparison, we also enable Cuda Graph for FP16 baseline.

## E Tuned Results on BERT

As mentioned in the main text and Appendix B.2, we use the same set of hyperparameters for BERT. However, tuning them can significantly boost the performance for ZeroQuant. Here, we tune two hyperparameters, i.e., the learning rate and the number of iterations in order to show the best possible performance of ZeroQuant

Table E.1: Result of  $\text{BERT}_{\text{base}}$  on the development set of GLUE benchmark (except WNLI). Here WxAy means x-bit for weight quantization and y-bit for activation quantization. Particularly, for W4/8, we quantize the MHSA’s weight to INT8 and FFC’s weight to INT4. Please see Appendix B.3 for the reported metrics.

Precision (Method)	CoLA	MNLI-m	MNLI-mm	MRPC	QNLI	QQP	RTE	SST-2	STS-B	Ave.
W32A32 (Baseline)	59.72	84.94	85.06	86.27/90.57	92.15	91.51/88.56	72.20	93.23	90.06/89.59	83.95
W8A8 (ZeroQuant-LKD No Tuning)	59.59	84.83	85.13	86.03/90.39	91.98	91.45/88.46	71.12	93.12	90.09/89.62	83.75
W8A8 (ZeroQuant-LKD Tuned)	60.90	84.95	85.10	86.27/90.60	92.07	91.47/88.47	71.84	93.46	90.09/89.62	84.07
W4/8A32 (ZeroQuant-LKD No Tuning)	58.50	83.16	83.69	84.80/89.31	90.83	88.94/84.12	70.04	92.78	88.49/88.67	82.35
W4/8A32 (ZeroQuant-LKD Tuned)	60.04	83.64	84.31	85.78/89.53	91.01	90.66/87.26	71.84	93.12	88.68/88.79	83.26
W4/8A8 (ZeroQuant-LKD No Tuning)	58.80	83.09	83.65	85.78/89.90	90.76	89.32/84.85	71.84	93.00	88.16/88.55	82.71
W4/8A8 (ZeroQuant-LKD Tuned)	60.30	83.47	84.03	85.78/89.90	90.87	90.77/87.38	71.84	93.00	88.38/88.70	83.22

Table E.2: Result of  $\text{BERT}_{\text{large}}$  on the development set of GLUE benchmark (except WNLI). Here WxAy means x-bit for weight quantization and y-bit for activation quantization. Particularly, for W4/8, we quantize the MHSA’s weight to INT8 and FFC’s weight to INT4. Please see Appendix B.3 for the reported metrics.

Precision (Method)	CoLA	MNLI-m	MNLI-mm	MRPC	QNLI	QQP	RTE	SST-2	STS-B	Ave.
W32A32 (Baseline)	63.35	86.65	85.91	87.99/91.62	92.24	91.08/88.08	74.01	93.46	90.34/90.11	85.03
W8A8 (ZeroQuant-LKD No Tuning)	63.38	86.52	85.64	87.75/91.50	92.31	91.09/88.05	72.56	93.35	90.45/90.19	84.81
W8A8 (ZeroQuant-LKD Tuned)	64.36	86.64	85.74	88.48/91.97	92.49	91.15/88.13	74.73	93.58	90.45/90.19	85.30
W4/8A32 (ZeroQuant-LKD No Tuning)	63.72	84.90	84.81	87.99/91.39	91.45	90.34/86.92	51.62	92.43	89.46/89.29	81.85
W4/8A32 (ZeroQuant-LKD Tuned)	64.06	85.02	84.98	88.73/91.99	91.82	90.45/87.12	52.35	92.78	89.72/89.44	82.19
W4/8A8 (ZeroQuant-LKD No Tuning)	63.51	84.70	84.71	88.73/91.99	91.73	90.25/86.74	49.82	92.09	89.34/89.08	81.62
W4/8A8 (ZeroQuant-LKD Tuned)	63.60	84.77	84.90	88.97/92.15	91.87	90.37/86.99	50.54	92.55	89.57/89.38	81.88

on both  $\text{BERT}_{\text{base}}$  and  $\text{BERT}_{\text{large}}$ . Particularly, we choose learning rate from the set {1e-6, 2e-6, 5e-6, 1e-5}, and choose number of iterations from the set {0, 50, 100, 200, 400, 800, 1600}. Thanks to the lightweight of LKD, the total tuning time for  $\text{BERT}_{\text{base}}$  (including all data loading time, evaluation time, tokenization time, all three quantization schemes, etc) is around 4.5 hours on 8 40G-A100 GPUs (i.e., 36 GPU hours), and the tuning time for  $\text{BERT}_{\text{large}}$  is around 16 hours on 8 40G-A100 GPUs (i.e., 128 GPU hours).

We summarize the best results in the Table E.1 and E.2.

## F QAT on $\text{BERT}_{\text{large}}$

We use four different learning rates for QAT on  $\text{BERT}_{\text{large}}$ , {5e-6, 1e-5, 2e-5, 5e-5}. The final results we reported in the paper are chosen from the best single run among those four different learning rates. However, even with such tuning, we are not able to get good performance for  $\text{BERT}_{\text{large}}$  on RTE.

Also, note that the time cost we used in the main text is based on a single run. if we consider the tuning cost, the total time will be  $4 \times 7181$ s

## G Limitations and Future Work

We believe it is critical for every work to clearly state its limitations, especially in this area. One limitation is that in this work we only focused on natural language models, but it would be interesting to see how ZeroQuant would perform for computer vision models. We leave this as a future work.

Another limitation is that we can only verify the scalability of ZeroQuant up to 20B scale models. If there are new releases of larger open-sourced models, it would be great to test ZeroQuant on those larger models as well.

Third, in this paper, we found out that the activation input of self-attention is more sensitive for quantization for the extra-large model (GPT-NeoX<sub>20B</sub>). However, we are unable to verify this on other extra-large models due to the lack of open-sourced models.

Table H.1: The full results of GPT-3<sub>350M</sub>.

Tasks	Baseline W32A32	PTQ				ZeroQuant			ZeroQuant-LKD	
		W8A32	W32A8	W8A8	W4/8A32	W8A8	W4/8A32	W4/8A8	W4/8A32	W4/8A8
HellaSwag	38.6	38.1	37.6	36.8	26.5	38.4	30.4	30.5	35.3	35.3
LAMBADA	49.3	49.3	44.7	42.9	0	51.0	10.1	10.5	39.8	37.4
TriviaQA	3.00	2.67	2.70	2.32	0	2.86	0.159	0.194	1.043	0.23
WebQs	1.43	0.935	1.23	0.689	0	1.378	0.246	0.394	0.591	0.049
Winogrande	53.2	52.1	52.1	52.1	47.8	51.4	52.6	50.7	51.6	51.8
PIQA	66.3	66.1	64.8	64.1	51.4	66.5	58.5	57.7	63.8	61.8
ARC (Challenge)	24.2	24.0	24.0	24.1	27.0	24.5	22.0	21.8	21.8	23.6
ARC (Easy)	45.5	44.7	44.2	43.9	25.1	44.5	37.6	37.5	40.5	40.5
ANLI R1	31.1	30.0	31.3	33.2	33.4	31.1	32.8	32.7	32.4	33.8
ANLI R2	34.3	36.0	36.5	35.9	33.4	34.3	34.7	34.2	34.1	33.5
ANLI R3	34.1	34.0	33.0	37.2	33.5	33.4	34.9	34.5	33.1	33.4
OpenBookQA	29.4	29.6	28.2	28.0	30.2	29.2	27.2	28.0	29.4	28.2
RACE-h	32.4	31.3	30.3	30.7	22.4	32.2	25.7	26.4	29.5	29.7
BoolQ	60.3	60.2	57.0	56.9	37.8	60.2	60.1	59.4	61.9	61.9
Copa	69.0	67.0	71.0	73.0	48.0	69.0	63.0	64.0	68.0	66.0
RTE	53.8	54.2	52.7	53.1	52.7	53.4	52.0	52.7	53.1	53.1
WSC	36.5	36.5	36.5	35.6	63.5	36.5	36.5	36.5	36.5	36.5
MultiRC	0.839	0.839	0.839	0.944	0.315	0.839	1.889	1.889	0.839	0.839
ReCoRD	75.1	74.8	69.2	67.5	16.1	74.9	56.5	55.3	70.1	68.5
Wikitext-2	21.52	22.09	24.56	26.20	1.76e5	21.68	88.64	92.10	30.56	31.13
Average Acc	38.86	38.54	37.78	37.84	28.9	38.71	33.52	33.42	37.02	36.64

## H Full Zero-shot Evaluation of GPT-3-style Models

We includes all zero-shot evaluation results in this section for all GPT-3-style models, inclunding GPT-NeoX<sub>20B</sub>.

Table H.2: The full results of GPT-3<sub>1.3B</sub>.

Tasks	Baseline W32A32	PTQ		ZeroQuant			ZeroQuant-LKD	
		W8A8	W4/8A32	W8A8	W4/8A32	W4/8A8	W4/8A32	W4/8A8
HellaSwag	51.4	47.0	26.1	50.8	43.7	43.2	48.5	46.7
LAMBADA	61.3	54.8	0	62.6	43.9	46.8	59.4	48.7
TriviaQA	7.37	4.43	0	6.67	2.36	2.09	4.28	2.99
WebQs	2.90	1.476	0	2.07	1.132	1.28	1.673	1.083
Winogrande	57.1	55.7	50.1	57.1	54.6	54.3	55.3	53.8
PIQA	71.4	67.7	50.4	70.7	66.5	66.4	69.5	68.1
ARC (Challenge)	27.2	27.1	26.5	26.8	25.7	25.3	27.8	26.5
ARC (Easy)	54.5	49.7	26.0	53.8	48.0	47.0	52.2	50.3
ANLI R1	32.0	33.1	33.0	33.4	33.8	33.6	34.2	33.8
ANLI R2	32.0	32.9	33.3	33.9	33.0	33.0	33.8	32.8
ANLI R3	33.8	33.5	32.3	34.8	33.6	33.5	33.7	33.0
OpenBookQA	33.6	32.6	27.0	33.4	30.0	28.8	31.6	29.0
RACE-h	33.6	32.6	22.4	32.7	30.9	29.9	32.7	33.2
BoolQ	62.4	59.2	37.8	61.3	60.3	59.8	61.7	61.3
Copa	70.0	70.0	55.0	72.0	73.0	74.0	72.0	70.0
RTE	53.1	54.5	50.9	52.7	52.7	52.7	52.7	52.0
WSC	37.5	36.5	63.5	36.5	36.5	36.5	36.5	36.5
MultiRC	1.05	0.839	0.315	0.839	1.259	1.154	0.839	0.839
ReCoRD	82.6	75.7	15.8	80.9	77.3	76.2	79.7	77.4
Wikitext-2	15.3	18.85	1.35e5	15.69	21.9	24.09	17.56	18.18
Average Acc	42.36	40.49	28.97	42.26	39.38	39.24	41.48	39.90

 Table H.3: The full results of W4/8A8 GPT-3<sub>350M</sub> using different data resources.

Tasks	Random Data	Wikipedia	Original Training Data
HellaSwag	33.9	35.5	35.3
LAMBADA	26.1	33.9	37.4
TriviaQA	0.088	0.972	0.23
WebQs	0.049	0.344	0.049
Winogrande	50.3	52.4	51.8
PIQA	59.3	62.4	61.8
ARC (Challenge)	22.6	23.3	23.6
ARC (Easy)	38.3	40.0	40.5
ANLI R1	33.0	32.0	33.8
ANLI R2	34.3	34.7	33.5
ANLI R3	33.4	32.9	33.4
OpenBookQA	29.2	28.0	28.2
RACE-h	27.8	29.1	29.7
BoolQ	47.8	52.6	61.9
Copa	65.0	69.0	66.0
RTE	50.5	52.7	53.1
WSC	36.5	36.5	36.5
MultiRC	1.574	1.154	0.839
ReCoRD	64.9	69.5	68.5
Wikitext-2	40.63	30.36	31.13
Average Acc	34.45	36.16	36.64

Table H.4: The full results of GPT-NeoX<sub>20B</sub>.

Tasks	W16A16	W8A8/16
HellaSwag	71.4	71.2
LAMBADA	71.7	71.9
TriviaQA	25.8	25.9
WebQs	6.3	6.64
Winogrande	66.0	65.7
PIQA	77.7	78.3
ARC (Challenge)	41.0	42.2
ARC (Easy)	68.5	68.8
ANLI R1	33.1	33.9
ANLI R2	33.4	34.4
ANLI R3	35.1	35.4
OpenBookQA	39.8	38.8
RACE-h	38.5	37.6
BoolQ	69.4	69.9
Copa	84.0	85.0
RTE	54.9	54.9
WSC	50.0	44.2
MultiRC	3.57	4.41
ReCoRD	88.3	88.0
Average Acc	50.45	50.38

# ZeroQuant-FP: A Leap Forward in LLMs Post-Training W4A8 Quantization Using Floating-Point Formats

Xiaoxia Wu\*, Zhewei Yao\*, Yuxiong He  
Microsoft

{zheweyao, xiaoxiauw, yuxhe}@microsoft.com

## Abstract

In the complex domain of large language models (LLMs), striking a balance between computational efficiency and maintaining model quality is a formidable challenge. Navigating the inherent limitations of uniform quantization, particularly when dealing with outliers, and motivated by the launch of NVIDIA’s H100 hardware, this study delves into the viability of floating-point (FP) quantization, particularly focusing on FP8 and FP4, as a potential solution. Our comprehensive investigation reveals that for LLMs, FP8 activation consistently outshines its integer (INT8) equivalent, with the performance edge becoming more noticeable in models possessing parameters beyond one billion. For weight quantization, our findings indicate that FP4 exhibits comparable, if not superior, performance to INT4, simplifying deployment on FP-supported hardware like H100. To mitigate the overhead from precision alignment caused by the disparity between weights and activations, we propose two scaling constraints for weight quantization that negligibly impact the performance compared to the standard W4A8 model. We additionally enhance our quantization methods by integrating the Low Rank Compensation (LoRC) strategy, yielding improvements especially in smaller models. The results of our investigation emphasize the immense potential of FP quantization for LLMs, paving the way for high-efficiency deployment in resource-limited settings.

## 1 Introduction

As Natural Language Processing (NLP) evolves, Large Language Models (LLMs) like Codex [9] and ChatGPT [22] have become essential, transforming our interaction with technology and daily communication. However, their complexity and computational intensity present deployment challenges [23, 8, 26], particularly in resource-limited settings. One solution is quantization, which represents data in lower-precision formats such as 8-bit integers or floating-point numbers, reducing memory needs and potentially enhancing inference latency through better GEMM computation throughput on compatible GPUs. Post-Training Quantization (PTQ), which directly reduces the precision of a fully trained model’s parameters, is often preferred for LLMs due to its simplicity and lower computational overhead.<sup>1</sup> Recent studies indicate that PTQ on 8-bit integer (INT8) weight-only quantization does not compromise the quality of LLMs [34, 3, 33, 29], and only a minor accuracy drop is observed with INT4 weight quantization when advanced algorithm such as GPTQ applied [7, 35, 12, 15].

The exploration of activation quantization, in addition to weight-only quantization, has also gained interest. This approach expedites inference times by taking advantage of unified precision leading to more efficient execution on hardware. The primary challenge in implementing activation quantization lies in the trade-off between efficiency and performance. As evidenced in studies such as ZeroQuants [34, 35], SmoothQuant [33] and others, reducing the precision of activation from FP16 to INT8 inevitably results in a decrease in model quality. This degradation is partially due to the presence of extreme values or outliers in the activation of

\*Equal Contribution. Code will be released as a part of <https://github.com/microsoft/DeepSpeed>

<sup>1</sup>Note that we do not discuss Quantize-Aware Training (QAT) for LLMs in this paper as QAT require the computation graph for back-propagation [11, 1, 18, 31, 32, 4, 16].

LLMs [5, 33, 15, 12], which is partly attributed to the pretraining effect [31]. In the presence of outliers, uniform quantization like INT8 or INT4, fail to accurately represent the main body of the data as they become skewed towards the outlier. This issue stems from the inherent assumption in these techniques of a uniform data distribution [30], an assumption that might not correspond to the actual data points distribution.

Considering the drawbacks of integer quantization delineated previously, floating-point (FP) methods like FP8 or FP4, employing ExMy notation, arise as more potent alternatives [20, 2, 13, 28, 37]. Unlike the fixed range of integer types, floating-point methods allow for adjusting the decimal point position, enabling dynamic scaling across activation maps and preserving important features. While there is debate about the quality of models between integer and floating-point quantization [28], recent research on PTQ LLMs using FP8/FP4 in [37] reveals FP8 to be substantially better than INT8 activation quantization. In terms of hardware support and performance, while INT8 computations are broadly supported by most modern CPUs and GPUs [21, 31], lower-bit floating-point operations are also increasingly recognized in the industry. An example of this is the newly release of NVIDIA’s H100 GPU, specifically engineered for FP8 computations [20]. Hence, despite the potentially higher computation cost of FP8 compared to INT8 and in light of hardware support, the improved model quality could make this trade-off worthwhile and merits further exploration.

While a few studies such as the one by [37] have ventured into the realm of post-training FP quantization in LLMs, they have unveiled considerable drawbacks in terms of model quality. Specifically, when implementing GPTQ [7] for FP8 quantization on both weights and activation for models such as LLaMA-7B or LLaMA-30b [27], there is an observed perplexity degradation surpassing 1.0 on Wiki-text2 dataset [19]. This level of model degradation presents significant practicality issues, hindering the optimal utilization of these models. In response to these findings, our paper undertakes an in-depth exploration into FP quantization. We primarily focus on the variance in activation values—an integral element that could potentially be the key to enhancing the performance of these quantization techniques. Our main contributions include:

- Demonstrating minimal degradation with FP8 activation and weight quantization: Particularly in larger models, FP8 activation and weight quantization result in negligible degradation, performing comparably to the original FP16 models.
- Identifying potential in FP8 activation and FP4 weights, and the impact of Low Rank Compensation (LoRC): We highlight the potential in FP8 activation and FP4 weights. The LoRC method, proposed in [35], significantly reduces quantization errors in the W4A8 scheme for FP quantization, especially in smaller models, thereby enhancing performance.
- Illustrating the maintenance of quality in the W4A8 floating-point model even when constraints are imposed on the scaling factors: For true efficiency in the W4A8 model, a conversion from FP4 to FP8 for weight is crucial. To alleviate this converting overhead, we here suggest two possible scaling constraints for weight quantization: (1) restricting all scaling factors to be a power of 2 and (2) requiring the scaling factors in one compute group (e.g., several rows of the weight matrix [34] to be transferable by simple bit-shifting). Our analysis indicates that these two restrictions negligibly affect the model’s performance in comparison to the conventional W4A8 configuration.

## 2 Background

The impact of 8-bit activation quantization, especially potential accuracy loss, is comprehensively outlined in ZeroQuant-V2 [35]. They present a direct comparison between the W16A16 and W16A8 (INT8) quantization schemes across a variety of models. To provide an easier understanding, we quoted their results in Table 1 for both OPT[36] and BLOOM [25] models, which indicates that the quality of models, especially the OPT family, is significantly influenced by the activation quantization.

Table 1: Comparison of FP16 and INT8 activation quantization. We report the average PPL (the lower the better) over Wikitext-2 (WIKI) [19], PTB [17], and C4 [24], for both OPT and BLOOM (BLM) models.

Precision	OPT-6.7b	OPT-13b	OPT-30b	OPT-66b	BLM-1.7b	BLM-3b	BLM-7.1b	BLM-176b
W16-A16	11.90	11.22	10.70	10.33	20.43	17.58	14.96	10.90
W16-A8	12.62	15.36	23.57	561.35	20.52	17.65	15.14	11.62

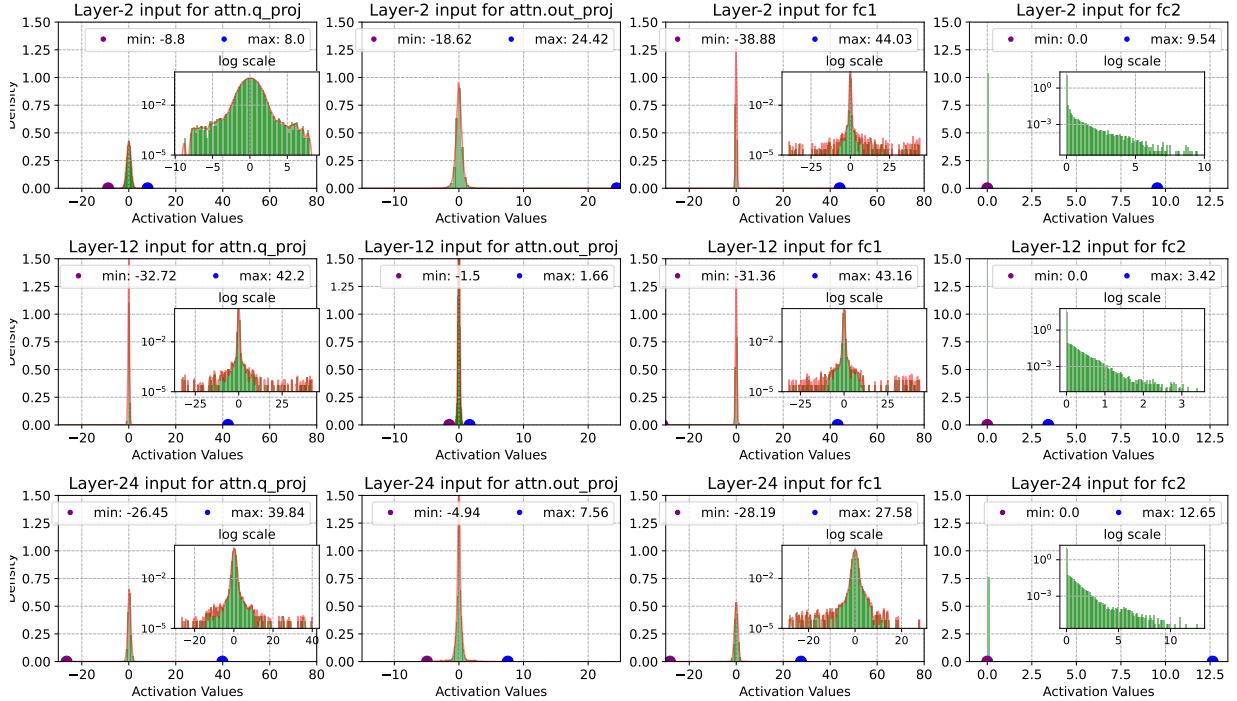


Figure 1: Distribution of Activation values. The top, middle and bottom rows represents the distributions at the 2nd, 12th and final layer of the pretrained OPT-1.3b model. From the left to right columns, they are respectively for the linear modules attn.q\_proj (same as attn.k\_proj and attn.v\_proj), attn.out\_proj, fc1, and fc2. The histogram’s x-axis ranges from the smallest to largest activation values, while the y-axis denotes their frequency in the dataset. See legend for their minimum and maximum values. Density functions illustrate the probability of different activation values. For more details, please see Section 2.

**Distribution of Activations.** We sought to understand the cause of the aforementioned degradation from FP16 activation and INT8, prompting us to scrutinize the distribution of activation values illustrated in Figure 1. We selected a random sentence from the C4 dataset and processed it through a pre-trained OPT-1.3B model. The statistical activation inputs for the 2nd, middle, and final layer were subsequently chosen for a detailed examination. The four histograms correspondingly represent the activations for the Multi-head Attention (MHA) and Multi-Layer Perceptron (MLP) components:<sup>2</sup>

- attn.q\_proj, the input for the query, key, or value in the MHA mechanism,
- attn.out\_proj, the input for the MHA’s projection matrices,
- fc1, the initial input for the fully-connected (fc1) projection in MLP,

<sup>2</sup>The hidden dimension for the model is 2048 for ‘attn.q\_proj’, ‘attn.out\_proj’ and ‘fc1’, and 8196 for ‘fc2’. We pick 20 tokens (position 8 to 28) and vectorize this  $20 \times 2048$  or  $20 \times 8196$  matrices to plot their distributions. The plots used bin=100.

- fc2, the subsequent input for the fully-connected (fc2) projection in the MLP.

The activation distribution outlined in Figure 1 reveals some compelling patterns. The input to the attn.q\_proj module in the 2nd layer (depicted in the 1st column of Figure 1 in the top row) appears to conform closely to a normal distribution, a result of the layer-normalization process. Yet, moving forward to the subsequent modules within the 2nd layer, namely attn.out\_proj, fc1, and fc2, we notice a skewness in the distributions, with noticeable outlier values. Two distinct observations arise: **(1)** Regarding the activation distribution for attn.q\_proj and fc1, even though they have undergone layer-normalization, the skewness still presents itself and becomes more conspicuous as we delve deeper into the layers (see the first and third column in the middle and bottom plots in Figure 1). **(2)** The skewness reaches its peak in the fc2 module. In this particular module, a large portion of the values cluster around zero, with only a handful surpassing this range. This phenomenon is due to the inputs being processed by the "ReLU" (Rectified Linear Unit) operator. This operator, purposefully, voids any negative input values, resulting in a skewed distribution focused around zero. Only positive activation values persist unmodified, giving rise to the outliers observed. This extreme skewness is most noticeable at the final layer (the bottom row in Figure 1).

These observations offer a deeper understanding of how activation quantization impacts various modules, even within the same layer. Consequently, this signifies that we must exercise caution when selecting quantization methods. Quantization techniques that employ integers, such as INT8 or INT4, and rely on uniform quantization, may not be ideally suited to manage distributions that are skewed. This is due to the inherent assumption of uniform distribution within these methods, which may not align with the actual distribution of data points.

**The Uniform Quantization of INT8.** The integer quantization such as in INT8 or INT4 states

$$Q(x) = \text{INT}((x - Z)/S) - Z, \quad (1)$$

where  $Q$  is the quantization function,  $x$  is a floating point input vector/tensor,  $S$  is a real valued scaling factor, and  $Z$  is an integer zero point. Based on different settings, the quantization method can be viewed as symmetric ( $Z = 0$ ) or asymmetric ( $Z \neq 0$ ) quantization. In scenarios where outliers exist, uniform quantization techniques like INT8 and INT4, regardless of their symmetric or asymmetric variants, frequently fail to accurately approximate the values of clustered data. Consequently, this makes the quantization error larger for those clustered values, as these methods attempt to adjust their fit to accommodate the outlier. Essentially, these techniques become skewed towards the outlier, leading to a reduced accuracy in representing the main body of the data.

Original	-0.4	-0.3	-0.2	-0.1	-0.001	0.0	0.001	0.1	0.2	0.3	0.4	0.5	1.0	10.0	100.0
INT8 Asymmetric Quantized	-0.394	-0.394	-0.394	0.0	0.0	0.0	0.0	0.0	0.394	0.394	0.394	0.394	1.181	9.843	100.006
FP8 (E5M2) Quantized	-0.375	-0.312	-0.188	-0.094	-0.001	0.0	0.001	0.094	0.188	0.312	0.375	0.5	1.0	10.0	96.0
FP8 (E4M3) Quantized	-0.406	-0.312	-0.203	-0.102	-0.002	0.0	0.002	0.102	0.203	0.312	0.406	0.5	1.0	10.0	104.0

Figure 2: A Contrast between INT8 and FP8 Quantization Methods. The top row displays the original vector in its full-precision form. The subsequent row showcases the vector after quantization through the INT8 Asymmetric approach. The final two rows present values quantized by the FP8 method, utilizing E5M2 and E4M3 formats respectively.

Given the limitations of integer quantization, floating-point methods such as FP8 or FP4, utilizing ExMy notation, emerge as superior alternatives. In these methods, the 'x' and 'y' values represent the bits allocated for the exponent and mantissa, respectively, totaling to 7 in FP8 or 3 in FP4. The flexibility of FP8 lies in its ability to adjust the decimal point position, unlike integer types with a fixed range.

To demonstrate the disparity between INT8 and FP8, we present Figure 2 where a hypothetical 15-element vector with an outlier value of 100 undergoes quantization using INT8 Asymmetric and FP8 (with both E5M2 and E4M3 configurations). Figure 2 illustrates that while INT8 approximates the outlier effectively, it struggles to accurately represent smaller numbers. Conversely, FP8 (whether with E5M2 or E4M3) provides greater precision in approximating the clustered data.<sup>3</sup>

Considering the advantages of ExMy, which allows for dynamic scaling across activation maps, quantization error is reduced and essential features are preserved. In this paper, we investigate the performance of FP8 or FP4 techniques for handling the variability in activation or weight values. This could potentially lead to an enhancement in the model’s performance on post-training quantization.

### 3 Methodology

Several lightweight optimization-based methods, where the weight of the model is updated during quantization, have been proposed in the literature [34, 35, 5, 33, 15, 12]. Among these, we chose to align our approach with the principles outlined in the GPTQ [7, 6], which can be dated back to [14, 10]. While this strategy offers a robust starting point, it is imperative to keep in mind the dynamic and ever-evolving nature of the field of artificial intelligence. There may be more efficient methodologies on the horizon, waiting to be discovered and implemented.

In light of ZeroQuant-V2 [35], we applied fine-grained quantization (FGQ) for weight and token-wise quantization for activation. In addition, we will also investigate the add-on feature LoRC (Low Rank Compensation) proposed in [35], which aims to reduce quantization errors in weights by employing low-rank matrix factorization. LoRC involves two main steps: first, it performs Singular Value Decomposition (SVD) on the error matrix, which is the difference between the original weight and the quantized weight. The error matrix is thus decomposed into two unitary matrices and a diagonal matrix. Second, the method formulates a new error approximation using two low-rank matrices that are derived from the matrices in the first step. This approximation is then added to the quantized weight to yield a more accurate estimate of the original weight, thereby reducing quantization errors.

Based on GPTQ (without or with LoRC), we perform comprehensive comparisons between the use of FP8 or INT8 activation quantization, coupled with adjusting the weight quantization to FP8 and FP4. Particularly we explore the potential of FP4 weight and FP8 activation quantization.

**Casting the FP4 to FP8.** Lastly, a unique challenge arises due to the use of different precision levels for weights (W) and activations (A). The actual software implementation of W4A8 in H100 NVIDIA hardware is that one needs to cast W’s FP4 to match the FP8 precision used in A. The direct method of dequantization followed by quantization again could potentially have a detrimental effect on inference efficiency, hence it is not a viable solution. To address this, we propose the bit-shifting method. This means that instead of allowing  $S$  defined in equation 1 to be any real valued scaling factor, we constrain  $S$  to be a power of 2, i.e.,  $S = 2^n, n \in N$  ( $S$  could still represent fractions when  $n$  is negative and whole numbers when  $n$  is not negative). There are two methods we will implement:

- (M1) Map to the nearest values represented by the power of 2, i.e., letting the new scale  $\hat{S} = 2^{\lceil \log_2(S) \rceil}$ ;
- (M2) Collect scales to form a vector  $\mathbf{S} = [S_1, S_2, \dots, S_n]$ . Take the maximum value in this group (usually, the set consists a (multiple) row(s) of a matrix [34]), denoted as  $S_{\max}$ , adjust these elements  $S_{\max}/S_i$  to be represented by the power of 2, and then define  $\hat{S}_i = S_{\max}/2^{\lceil \log_2(S_{\max}/S_i) \rceil}$ . This provides a far superior approximation compared to (M1).<sup>4</sup>

We reiterate that this restriction using the power of 2, either using (M1) or (M2), simplifies computations, especially in digital systems operating based on binary logic. This is a crucial element of our approach to optimizing computational efficiency and maintaining the performance of our model.

---

<sup>3</sup>Please note that the FP8 format used in this paper is based on the Qtorch Python package, which can be installed via ‘pip install qtorch’. It differs slightly from Nvidia’s FP8 in H100, which requires one mantissa bit-pattern for NaN values.

<sup>4</sup>To ensure the casting of F4-E2M1 for each weight matrix to FP8, we apply format E5M2 once a matrix is quantized.

Table 2: The evaluation outcomes for LLaMA (top) and OPT (bottom) using different Integer (INT) and Floating-point (FP) quantization methods applied to weight and activation. The performance is measured in terms of perplexity (lower scores are better) and spans across three datasets: WikiText-2 (WIKI), PTB, and C4. For each model, the results initially highlight the average performance across the datasets, followed by a detailed breakdown of outcomes per dataset.

Q-type	Weight-Activation	LLaMA-3b			LLaMA-7b			LLaMA-13b			LLaMA-30b			
		Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	
W16A16	N/A	11.93	7.35/19.1/9.34	13.37	5.68/27.35/7.78	10.31	5.09/19.22/6.61	5.79	4.10/7.30/5.98					
W8A8	INT – INT	12.00	7.41/19.16/9.41	13.58	5.72/27.89/7.13	10.63	5.16/20.07/6.67	5.90	4.21/7.42/6.06					
	INT – FP	11.96	7.37/19.16/9.35	13.45	5.69/27.57/7.09	10.38	5.11/19.42/6.62	5.80	4.11/7.31/5.99					
	FP – FP	11.99	7.37/19.23/9.37	13.46	5.70/27.58/7.10	10.38	5.11/19.41/6.62	5.81	4.12/7.31/5.99					
W4A8	INT – INT	12.55	7.67/20.23/9.74	16.23	6.44/34.45/7.79	11.48	5.32/22.35/6.78	6.02	4.36/7.54/6.16					
	INT – FP	12.39	7.62/19.87/9.68	16.09	6.75/33.80/7.72	11.31	5.28/21.91/6.73	5.94	4.27/7.45/6.11					
	FP – FP	12.45	7.62/20.05/9.67	15.14	6.32/31.61/7.51	11.08	5.26/21.27/6.73	5.92	4.26/7.42/6.09					
W4A8 +LoRC	INT – INT	12.52	7.65/20.18/9.72	14.14	5.88/29.26/7.27	10.81	5.28/20.38/6.76	6.00	4.34/7.51/6.14					
	INT – FP	12.38	7.58/19.89/9.65	14.01	5.84/28.95/7.24	10.56	5.22/19.75/6.71	5.90	4.24/7.39/6.07					
	FP – FP	12.42	7.61/19.98/9.66	13.95	5.87/28.75/7.24	10.80	5.24/20.46/6.72	5.91	4.26/7.40/6.07					
Q-type	Weight – Activation	OPT-3b			OPT-7b			OPT-13b			OPT-30b			
		Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	
W16A16	N/A	15.44	14.62/16.97/14.72	11.90	10.86/13.09/11.74	11.22	10.13/12.34/11.20	10.70	9.56/11.84/10.69					
W8A8	INT – INT	15.94	14.98/17.49/15.36	12.66	11.20/14.29/12.48	15.94	12.13/19.82/15.86	25.76	14.63/32.90/29.74					
	INT – FP	15.85	14.93/17.56/15.05	11.99	10.92/13.24/11.80	11.27	10.16/12.42/11.23	10.69	9.51/11.87/10.71					
	FP – FP	15.86	14.97/17.55/15.05	11.99	10.91/13.24/11.81	11.27	10.16/12.42/11.23	10.69	9.51/11.87/10.71					
W4A8	INT – INT	16.41	15.39/18.22/15.62	13.18	11.61/15.00/12.92	16.70	12.32/21.21/16.56	24.42	14.80/30.38/28.09					
	INT – FP	16.40	15.46/18.23/15.51	12.20	11.13/13.49/11.99	11.34	10.20/12.53/11.30	10.73	9.54/11.91/10.75					
	FP – FP	16.29	15.32/18.19/15.35	12.09	10.89/13.44/11.95	11.34	10.16/12.55/11.30	10.72	9.52/11.90/10.75					
W4A8 +LoRC	INT – INT	16.38	15.50/18.05/15.59	12.75	11.37/14.33/12.53	15.89	12.06/19.76/15.85	27.20	15.94/34.50/31.16					
	INT – FP	16.23	15.40/17.97/15.32	12.13	11.07/13.43/11.90	11.34	10.23/12.49/11.29	10.71	9.48/11.91/10.74					
	FP – FP	16.23	15.50/17.92/15.28	12.09	10.96/13.40/11.90	11.33	10.15/12.55/11.29	10.71	9.48/11.90/10.75					

## 4 Main Results

In this section, we perform experiments to understand the differences of Integer (INT) and Floating-point (FP) quantization using the GPTQ methods [7] with or without the add-on feature LoRC [35]. As described in Section 2, floating-point quantization could potentially maintain more precise information, which might improve the model’s performance. To see if this is true, we include two model-type families: LLaMA [27] and OPT [36], with sizes ranging from 1 billion to 30 billion parameters. The evaluation spans across three datasets: Wikitext-2 (WIKI) [19], PTB [17], and C4 [24]. For more experiment details, please see Appendix A.

The primary results in Table 2 reveal the impact of various quantization types which are applied to weight and activation specified in the 2nd column; for instance, W4A8 precision, INT – FP means INT4 is used for weight and FP8 for activation. Our results provide an average performance over three datasets, offering a broad understanding of the quantization method’s efficiency. However, we delve further, understanding that these methods’ performance can differ with the characteristics of datasets, thus presenting a detailed performance breakdown for each dataset. We find that FP8 and FP4, the configurations E4M3 and E2M1 respectively outperform E5M2 and E3M0, hence, they were used in our experiments. Further insights and explanations regarding these configurations’ impact on performance are to be addressed in Appendix A.

**FP8 Activation is much better than INT8.** The high-level summary of the results in Table 2 indicates that for both LLaMA and OPT model families, FP8 activation generally outperforms INT8 activation. This observation corroborates the motivation discussed in Section 2, emphasizing FP8’s superior capacity to capture more nuanced information, a vital aspect for generative tasks in large-scale LLMs.

Interestingly, the advantage of FP8 over INT8 becomes more pronounced for larger models with parameters greater than 6.7 billion, such as LLaMA-7b/13b and OPT-6.7b/13b. For instance, when considering LLaMA-

7b, shifting from INT to FP quantization in the W8A8 configuration leads to an additional 0.25 PPL reduction (from 10.63 to 10.38), and in the W4A8 setup, there is an extra 0.4 PPL drop (from 11.48 to 11.08). These performance gains are significant, considering all other optimization parameters remain constant, and they align with the Class-3 quantization sensitivity category as defined in [35]. Thus, the results underline the importance of FP8 activation, particularly in larger LLMs, to enhance the overall performance and precision of the model’s outputs.

**FP8 weights rival INT8, while FP4 weights potentially outperform INT4.** From Table 2, we observe comparable performances between INT8 and FP8 weight quantization across various models and datasets, when keeping activation at FP8. This probably due to we used FGQ on weight quantization. Interestingly, when weight quantization is lowered, FP4 exhibits certain advantages over INT4, particularly evident in LLaMA-7b (15.14 to 16.09) and LLaMA-13b models (11.08 to 11.31). Specifically, under the W4A8 configuration for LLaMA-7b, we see 0.95 improvement of FP4 over INT4, a significant gain. The preferable performance of FP4 over INT4 is particularly advantageous for hardware designs like H100, where FP8 is already supported. Thus, a simple modification to accommodate FP4 would be easier than implementing a system supporting INT4 weight and FP8 activation.

**LoRC improves W4A8.** Table 2 shows that the Low Rank Compensation (LoRC) method enhanced the W4A8 quantization scheme, reducing quantization errors. This improvement is particularly pronounced in smaller models, underlining the effectiveness of LoRC in optimizing the performance of these computing processes while impacting little on the model-size.

Table 3: Scale values ( $S$ ) are evaluated both without and with restrictions of being a power of 2, as shown in the second column. The quantization type employed is FP4 for weight and FP8 for activation.

Q-type	Scale $S = 2^n$	LLaMA-3b		LLaMA-7b		LLaMA-13b		LLaMA-30b	
		Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4
W4A8	✗	12.45	7.62/20.05/9.67	15.14	6.32/31.61/7.51	11.08	5.26/21.27/6.73	5.92	4.26/7.42/6.09
	✓(M1)	12.66	7.76/20.41/9.81	16.33	6.34/34.82/7.82	10.90	5.31/20.63/6.78	6.00	4.38/7.48/6.15
	✓(M2)	12.55	7.68/20.21/9.77	14.49	6.37/29.32/7.79	10.95	5.26/20.81/6.77	5.97	4.31/7.48/6.13
W4A8 LoRC	✗	12.42	7.61/19.98/9.66	13.95	5.87/28.75/7.24	10.80	5.24/20.46/6.72	5.91	4.26/7.40/6.07
	✓(M1)	12.61	7.69/20.37/9.76	14.23	5.94/29.47/7.30	10.74	5.28/20.19/6.75	5.98	4.33/7.48/6.13
	✓(M2)	12.42	7.63/19.89/9.74	13.68	5.90/27.83/7.32	10.40	5.23/19.22/6.76	5.94	4.28/7.44/6.11
Q-type	Scale $S = 2^n$	OPT-1.3b		OPT-6.7b		OPT-13b		OPT-30b	
		Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4	Mean	WIKI/PTB/C4
W4A8	✗	16.29	15.32/18.19/15.35	12.09	10.89/13.44/11.95	11.34	10.16/12.55/11.30	10.72	9.52/11.90/10.75
	✓(M1)	16.66	15.65/18.66/15.65	12.29	11.12/13.69/12.05	11.36	10.22/12.54/11.32	10.77	9.58/11.96/10.76
	✓(M2)	16.47	15.23/18.55/15.62	12.25	11.11/13.61/12.03	11.40	10.22/12.61/11.36	10.74	9.47/11.96/10.78
W4A8 LoRC	✗	16.23	15.50/17.92/15.28	12.09	10.96/13.40/11.90	11.33	10.15/12.55/11.29	10.71	9.48/11.90/10.75
	✓(M1)	16.47	15.59/18.37/15.45	12.17	11.10/13.47/11.95	11.36	10.21/12.54/11.32	10.74	9.49/11.96/10.76
	✓(M2)	16.30	15.39/18.10/15.42	12.19	11.11/13.49/11.97	11.41	10.34/12.54/11.34	10.75	9.49/11.96/10.78

**Casting the FP4 to FP8.** As detailed in Section 3, to maximize real latency speedup on NVIDIA H100 hardware, we suggest the scale factor  $S$  for weight quantization to be represented as a power of 2. In pursuit of this, we executed a series of experiments using FP4 for weight and FP8 for activation quantization. The results of these experiments, conducted both with and without LoRC, are presented in Table 3. Our data shows that while constraining the scaling factors occasionally results in unexpected improvements in models like LLaMA-7b and LLaMA-13b, we generally observe a minor degradation of quality in the W4A8 floating-point model, regardless of whether we used method M1 or M2. M2 generally outperforms M1. When we implement LoRC, this decline in quality can be mitigated, particularly in the OPT-1.3b, LLaMA-7b, and LLaMA-13b models. Hence, our results advocate for the use of LoRC, especially when considering scale restrictions for weight quantization in deep learning models.

## 5 Conclusions

In this study, we demonstrate that floating-point (FP) quantization significantly surpasses integer (INT) quantization in the context of large language models (LLMs) during post-training quantization. Notably, FP8 activation exceeds INT8, especially in larger models. Moreover, FP8 and FP4 weight quantization are either competitive with or surpass their INT equivalents. The Low Rank Compensation (LoRC) approach greatly enhances the W4A8 quantization scheme, particularly in smaller models. In conclusion, our work underscores the potential of FP quantization in enhancing model performance, and strategies such as LoRC further mitigate degradation induced by scale factor restrictions on weight quantization.

## Acknowledgement

This research was conducted within the supportive environment of the DeepSpeed team at Microsoft, whose invaluable assistance was instrumental to this project. We thank Cheng Li and Connor Homes for the insightful discussions.

## References

- [1] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. *Advances in neural information processing systems*, 31, 2018.
- [2] Léopold Cambier, Anahita Bhiwandiwalla, Ting Gong, Mehran Nekuii, Oguz H Elibol, and Hanlin Tang. Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks. *arXiv preprint arXiv:2001.05674*, 2020.
- [3] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.
- [4] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.
- [5] Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. *arXiv preprint arXiv:2212.09720*, 2022.
- [6] Elias Frantar and Dan Alistarh. Optimal brain compression: A framework for accurate post-training quantization and pruning. *arXiv preprint arXiv:2208.11580*, 2022.
- [7] Elias Frantar, Saleh Ashkboos, Torsten Hoefer, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.
- [8] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W Mahoney, and Kurt Keutzer. Ai and memory wall. *RiseLab Medium Post*, 2021.
- [9] GitHub. Github copilot. <https://github.com/features/copilot/>, 2021.
- [10] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171, 1993.
- [11] Minje Kim and Paris Smaragdis. Bitwise neural networks. *arXiv preprint arXiv:1601.06071*, 2016.
- [12] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*, 2023.
- [13] Andrey Kuzmin, Mart Van Baalen, Yuwei Ren, Markus Nagel, Jorn Peters, and Tijmen Blankevoort. Fp8 quantization: The power of the exponent. *arXiv preprint arXiv:2208.09225*, 2022.

- [14] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.
- [15] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.
- [16] Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. Llm-qat: Data-free quantization aware training for large language models. *arXiv preprint arXiv:2305.17888*, 2023.
- [17] Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Using Large Corpora*, page 273, 1994.
- [18] Naveen Mellempudi, Abhisek Kundu, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. Ternary neural networks with fine-grained quantization. *arXiv preprint arXiv:1705.01462*, 2017.
- [19] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017.
- [20] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al. Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.
- [21] NVIDIA. FasterTransformer. <https://github.com/NVIDIA/FasterTransformer>, January 2023.
- [22] OpenAI. Openai chatgpt. <https://openai.com/blog/chatgpt/>, 2022.
- [23] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*, 2022.
- [24] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019.
- [25] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.
- [26] Shaden Smith, Mostafa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.
- [27] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- [28] Mart van Baalen, Andrey Kuzmin, Suparna S Nair, Yuwei Ren, Eric Mahurin, Chirag Patel, Sundar Subramanian, Sanghyuk Lee, Markus Nagel, Joseph Soriaga, et al. Fp8 versus int8 for efficient deep learning inference. *arXiv preprint arXiv:2303.17951*, 2023.
- [29] Xiuying Wei, Yunchen Zhang, Yuhang Li, Xiangguo Zhang, Ruihao Gong, Jinyang Guo, and Xianglong Liu. Outlier suppression+: Accurate quantization of large language models by equivalent and optimal shifting and scaling. *arXiv preprint arXiv:2304.09145*, 2023.
- [30] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602*, 2020.

- [31] Xiaoxia Wu, Cheng Li, Reza Yazdani Aminabadi, Zhewei Yao, and Yuxiong He. Understanding int4 quantization for transformer models: Latency speedup, composability, and failure cases. *arXiv preprint arXiv:2301.12017*, 2023.
- [32] Xiaoxia Wu, Zhewei Yao, Minjia Zhang, Conglong Li, and Yuxiong He. Extreme compression for pre-trained transformers made simple and efficient. *arXiv preprint arXiv:2206.01859*, 2022.
- [33] Guangxuan Xiao, Ji Lin, Mickael Seznec, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*, 2022.
- [34] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *arXiv preprint arXiv:2206.01861*, 2022.
- [35] Zhewei Yao, Cheng Li, Xiaoxia Wu, Stephen Youn, and Yuxiong He. A comprehensive study on post-training quantization for large language models. *arXiv preprint arXiv:2303.08302*, 2023.
- [36] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuhui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.
- [37] Yijia Zhang, Lingran Zhao, Shijie Cao, Wenqiang Wang, Ting Cao, Fan Yang, Mao Yang, Shanghang Zhang, and Ningyi Xu. Integer or floating point? new outlooks for low-bit quantization on large language models. *arXiv preprint arXiv:2305.12356*, 2023.

## A Experiment Details

As we used GPTQ method [7], we use C4 dataset to randomly select 128 sentences for the light-weight PTQ and each of them has 2048 tokens. We run them on a single GPU (i.e, V100-32GB) thanks for the two open-source github repositories.<sup>5</sup> To accommodate the real computation efficiency, the group-size for weight quantization is 256 for both model family (OPT and LLaMA) except the LLaMA-3b with 320 as its hidden-dimension is 3200. All the checkpoints we used are from huggingface.<sup>6</sup> As for activation, we perform token-wise quantization in order to accommodate the latency requirements. For LoRC method, the dimension for the two low-rank matrix we used for LLaMA is 8. While for OPT, the dimension is 16, 32, 40 and 56 respectively for 1.3b, 6.7b, 13b and 30b. We did not try others dimension as indicated by [35] that dimension of the low-rank matrix does not play too much impact on the quantization error as long as it larger than 8.

Table A.1: Comparisons between E2M1 and E3M0. The quantization is FP4 for weight and FP8 for activation.

Activation (FP8)	OPT-1.3b	OPT-6.7b	OPT-13b	OPT-30b
Weight-FP4 (E3M0)	16.96	12.41	11.53	10.86
Weight-FP4 (E2M1)	16.23	12.09	11.33	10.71

<sup>5</sup><https://github.com/IST-DASLab/gptq> and <https://github.com/qwopqwop200/GPTQ-for-LLaMa.git>

<sup>6</sup>LLaMA-3b is [openlm-research/open\\_llama\\_3b](https://openlm-research.github.io/llama_3b/) and all other LLaMA are from [decapoda-research/llama-#b-hf](https://decapoda-research.com/llama-#b-hf) where # can be 7b, 13b and 30b. As for OPT, they are from [facebook/opt-#b](https://facebookresearch.com/opt-#b) where # can be 1.3b, 7b, 13b and 30b.

# ZeroQuant-V2: Exploring Post-training Quantization in LLMs from Comprehensive Study to Low Rank Compensation

Zhewei Yao\*, Xiaoxia Wu\*, Cheng Li, Stephen Youn, Yuxiong He  
Microsoft

{zheweyao, xiaoxiawu, chengli1, stephen.youn, yuxhe}@microsoft.com

## Abstract

Post-training quantization (PTQ) has emerged as a promising technique for mitigating memory consumption and computational costs in large language models (LLMs). However, a systematic examination of various quantization schemes, model families, and quantization bit precision has been absent from the literature. In this paper, we conduct a comprehensive analysis of these factors by investigating the effects of PTQ on weight-only, activation-only, and weight-and-activation quantization using diverse methods such as round-to-nearest (RTN), GPTQ, ZeroQuant, and their variants. We apply these methods to two distinct model families with parameters ranging from 125M to 176B. Our contributions include: (1) a sensitivity analysis revealing that activation quantization is generally more susceptible to weight quantization, with smaller models often outperforming larger models in terms of activation quantization; (2) an evaluation and comparison of existing PTQ methods to optimize model size reduction while minimizing the impact on accuracy, revealing that none of the current methods can achieve the original model quality for quantization with either INT4-weight or INT4-weight-and-INT8-activation; (3) based on these insights, we propose an optimized method called Low-Rank Compensation (LoRC), which employs low-rank matrices to enhance model quality recovery with a minimal increase in model size.

## 1 Introduction

Large language models (LLMs) like Codex [15] and ChatGPT [24] have demonstrated breakthrough performance across various benchmarks, such as natural language understanding and generation, and are now integrated into everyday applications. However, efficiently serving LLMs has become a pressing concern due to their significant memory consumption and computational demands. Unlike classification or diffusion models, LLMs present unique challenges, as they involve two distinct phases: prompt and generation. The prompt phase is primarily compute-bound, while the generation phase, with low batch size and KV cache, is mainly memory-bound [26].

As the progression of hardware bandwidth lags behind that of computational demand [14], the resource demands of extra-large models such as MT-NLG-530B [30]—which necessitates the deployment of multiple nodes for operation—escalate, adding to the complexities of cross-node communication. This has emphasized the urgency to curtail both the size and computational expense of Large Language Models (LLMs). An increasingly effective solution to these issues is post-training quantization (PTQ). This method aids in the reduction of training prerequisites while simultaneously lowering the bit precision of weights and activations to either INT4 or INT8.

While the effectiveness of post-training quantization (PTQ) has been underscored in a number of recent studies [36, 12, 35, 7], a comprehensive, systematic investigation into several key dimensions of this technique remains to be undertaken. Specifically, the extant literature falls short in providing thorough coverage of the functionality of various PTQ methods or the sensitivity of disparate models. Moreover, despite current quantization methods demonstrating promising results in the reduction of model sizes, the question persists

---

\*Equal Contribution. Code will be released as a part of <https://github.com/microsoft/DeepSpeed>

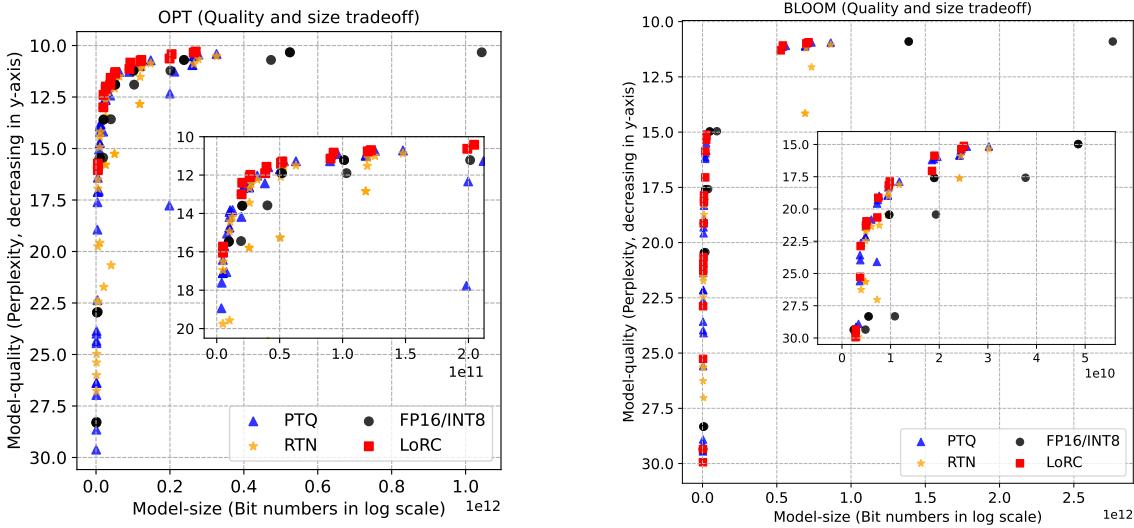


Figure 1: The model size and quality trade-off of different quantization methods on models from OPT and BLOOM families. Here PTQ (with fine-grained quantization) represents the method from [36, 12], RTN means the naive round-to-nearest baseline (with fine-grained quantization as well), and FP16/INT8 is used as the no-accuracy-loss baseline. LoRC is our proposed method that works seamless with PTQ. Note that we drop all diverged points for better visualization. For all detailed numbers, please see Appendix E.

as to whether these methods are achieving their optimal potential in minimizing Large Language Models (LLMs) sizes.

With these observations in mind, our study sets forth to address two salient questions: (1) When subjected to quantization, do LLMs of varying sizes and pretraining data exhibit similar behavior? (2) Are existing quantization methods truly leveraging their full potential in reducing the sizes of LLMs?

**Contribution.** To elucidate these queries, we undertake an exhaustive examination of the impact of PTQ on weight-only, activation-only, and combined weight-and-activation quantization. This investigation incorporates a range of PTQ methods, including round-to-nearest (RTN), GPTQ [12], ZeroQuant [36], and their respective variants. To broaden the scope of our analysis, we focus on two distinct model families, OPT [40] and BLOOM [28], spanning model sizes from 125M to a massive 176B. Our code will be made available for reproduction. In summary, we make the following contributions:

(1) We provide a thorough **sensitivity analysis** to demonstrate that a) Activation quantization is generally more sensitive to weight quantization; Smaller models usually have better activation quantization performance than the relative larger model. b) Different model families show different INT8 activation quantization behaviors; Particularly for large models, BLOOM-176B has small accuracy drops (about 1 perplexity or PPL) but OPT-30B and -66B experience worse performance.

(2) We carry out a detailed evaluation and comparison of current PTQ methods, utilizing optimal configurations to maximize model size reduction while minimizing accuracy impact. We found that the current existing method can barely achieve less than 0.1 PPL points degradation for quantization with either INT4-weight or INT4-weight-and-INT8-activation (W4A8). To recover the 0.1 PPL, we strive to push the boundaries of employing **fine-grained quantization** (FGQ) techniques. We observe FGQ is able to recovered points degradation of <0.1 PPL for large models (>13B) for INT4 weight quantization, but there are still non-negligible model quality drops.

(3) Based on the above understanding, we further optimize existing methods and introduce a technique called **Low Rank Compensation** (LoRC), which employs low-rank matrix factorization on the quantization error matrix. Complementary to FGQ, LoRC plays a crucial role in enhancing the full model quality recovery,

while there is little increase of the model size.

In Figure 1, we provide model size and quality trade-offs for both OPT and BLOOM families. As can be seen, using LoRC on top of PTQ methods from [36, 12] and fine-grained quantization, we set a new quantization Pareto frontier for LLMs. Meanwhile, we recommend the following setting for quantizing LLMs with LoRC (Note that activation quantization should be only applied if necessary): (1) For larger models ( $>10B$ ), fine-grained (block size 64–256) 4-bit weight quantization plus 8-bit activation quantization (block size 64–256) with PTQ can be used for real deployment; (2) For middle-size models ( $<10B$  and  $>1B$ ), per-row INT8 quantization plus fine-grained (block size 64–256) INT8 activation quantization can be used with PTQ from [12, 36]; (3) For smaller models ( $<1B$ ), per-row W8A8 (INT8 weight and INT8 activation) RTN is enough based on [36].

## 2 Related Work

Different quantization methods [29, 38, 9, 41, 1, 8, 31, 19] for transformer-based models [32] have been explored for a while. However, most of those works need quantization-aware finetuning or even expensive quantization-aware knowledge distillation [17]. Due to the cost of training/finetuning LLMs [25, 18, 31, 34, 33], it is a challenge for practitioners/researchers to do finetuning/distillation on those LLMs, particularly for models like GPT-3-175B [4] and BLOOM-176B [28].

Post-training quantization (PTQ) [37, 3] is an alternative way to quantize the model with no/minimal finetuning requirement. Along this line, several recent works focus on LLMs (beyond the million-parameter scale). [36] proposes vector-based INT8 quantization with layer-by-layer knowledge distillation to overcome the training cost and quantization error introduced by LLMs. [6] uses similar vector-based INT8 quantization weight plus mixed-precision (INT8/FP16) quantization for activation to overcome the sensitivity of activation quantization. However, the inference speed of [6] is generally even slower than FP16 baseline [2] due to the difficulty of implementing mixed-precision calculation within a single tensor. More recently, [12] extends OBQ [10, 16, 21] on LLMs for INT4 weight-only quantization and shows great efficiency on quantization and latency, and [35] shows the outliers from activations can be smoothed out by migrating the quantization difficulty from activations to its associated weights. However, [35] can only work for W8A8 quantization as lower weight precision (INT4) itself already leads to significant accuracy degradation, and the accuracy drop is larger than 0.1 PPL points, which as discussed in the later section is sub-optimal. [7] shows the scaling law of weight-only quantization with the simplest round-to-nearest baseline, but it does not consider the weight-and-activation quantization and/or the above PTQ optimization methods. As can be seen from Figure 1, by using PTQ optimization methods, the model quality can be significantly improved. Please also see Appendix E for more detailed numbers.

Different than existing works, our paper extensively tests the effect of (1) different quantization schemes, e.g., symmetric and asymmetric quantization, (2) different PTQ methods, e.g., [36, 12], (3) different model families, e.g., [28, 40], (4) different quantization coverage, e.g., weight-only and weight-and-activation quantization, and (5) other discussions, e.g., the effect of quantization granularity. As such, we provide a much more comprehensive understanding of post-training quantization for large language models compared to the previous works.

## 3 Would different model families behave similarly on quantization?

There are mainly two categories of PTQ for LLMs, i.e., weight-only quantization [12] and weight-and-activation quantization [6, 36, 35]. In the latter, it is uniformly observed across all studies that activation quantization demonstrates greater sensitivity than weight quantization. However, prior research tends to concentrate on a single (family) model to emphasize the necessity of their proposed quantization technique. A comprehensive and systematic evaluation of this PTQ methodology, particularly the sensitivity of weight/activation quantization for varying model sizes and distinct model families, has yet to be undertaken. Hence, we conduct an examination on both the OPT [40] and BLOOM [28] families to elucidate the quantization sensitivity of

weight and activation.

**Sensitivity setting.** We use the zero-shot validation perplexity (PPL) differential on three datasets, namely, WikiText-2 [23], PTB [22], and C4 [27], before and after the quantization of these LLMs to illustrate their sensitivity, as PPL is significantly correlated to zero-shot/few-shot accuracy measurement [7]. Specifically, a higher PPL drop indicates enhanced quantization sensitivity. For simplicity, we also categorize quantization sensitivity (or quantization loss) into three different classes as depicted in Table 1. Notably, the threshold is chosen because when the model size approximately doubles (e.g., 13B vs. 30B, and 30B vs. 66B), the PPL improvement is about 0.5 (see Table 2). The sensitivity (or loss) incrementally increases as the class number ascends. From a practical standpoint, we favor lower quantization sensitivity (accuracy loss), making **Class-1** the optimal-loss post-training quantization.

We employ both symmetric and asymmetric quantization to gauge the quantization sensitivity and highlight the advantage of asymmetric quantization. Particularly, we implement per-row quantization [12] for weight quantization and per-token quantization for activation [36].

**Robustness of Weight-only Quantization for Large Models.** The results of weight-only quantization in OPT and BLOOM models are summarized in Table 2. INT8 weight-only quantization, either symmetric or asymmetric, results in negligible accuracy loss (less than 0.05, i.e., **Class-1**). Consequently, for tasks oriented towards generation, FP16 weight can simply be replaced with INT8 weight to reduce memory usage. For INT4 quantization, the asymmetric method outperforms the symmetric approach in accuracy, attributable to its superior utilization of the quantization range. Interestingly, larger models exhibit better tolerance to low-precision quantization (i.e., INT4) than smaller models, with a few exceptions such as OPT-66B.<sup>1</sup> Particularly, BLOOM-176B shows PPL degradation (around 0.3 points) in **Class-2**, which could explain why the large GLM-130B [39] can operate with INT4 weight-only quantization out of the box with acceptable accuracy impact.

Table 2: Average PPL of OPT and BLOOM (BLM). See Table E.1 for all results.

Precision	OPT-6.7b	OPT-13b	OPT-30b	OPT-66b	BLM-1.7b	BLM-3b	BLM-7.1b	BLM-176b
W16-A16	11.90	11.22	10.70	10.33	20.43	17.58	14.96	10.90
W8 <sup>sym</sup> -A16	11.90	11.22	10.70	10.33	20.43	17.59	14.97	10.90
W8 <sup>asym</sup> -A16	11.90	11.22	10.70	10.33	20.45	17.59	14.97	10.90
W4 <sup>sym</sup> -A16	14.36	12.73	11.77	97.05	23.18	19.36	16.27	11.28
W4 <sup>asym</sup> -A16	13.44	12.09	11.52	31.52	22.47	19.01	15.90	11.20
W16-A8 <sup>sym</sup>	26.04	3171.49	2048.21	2638.09	20.68	17.73	15.28	12.10
W16-A8 <sup>asym</sup>	12.62	15.36	23.57	561.35	20.52	17.65	15.14	11.62

**Challenge Encountered in Activation Quantization for Large Models.** Activation quantization has consistently proven more difficult than weight quantization [36, 6], as illustrated in Table 2. When compared to weight-only quantization, activation-only quantization indicates that asymmetric quantization can significantly improved performance over symmetric quantization. Moreover, contrary to weight-only quantization, smaller models typically exhibit better tolerance to activation quantization, as their hidden dimension is smaller and the activation dynamic range is also narrower than larger models [36]. It should be

<sup>1</sup>[12] discovered that OPT-66B has a high proportion of dead neurons in the early layers, which might influence the compression capability. We also identify another potential reason: the Layer Norm of the OPT-family is not well trained (except OPT-350M), with the weight and the bias being all 1’s and 0’s, respectively.

Table 1: Classification of quantization sensitivity (or quantization loss). The sensitivity increases from **Class-1** to **Class-3**.

Class	<b>Class-1</b>	<b>Class-2</b>	<b>Class-3</b>
PPL Degradation	$\leq 0.1$	$> 0.1 \text{ & } \leq 0.5$	$> 0.5$

noted that for models larger than 10B, all fall into *Class-3*, indicating a degradation of more than 0.5 PPL points.

The last two rows of Table 2 show that different model families exhibit significantly different behaviors. BLOOM does not exhibit divergence issues even up to a model size of 176B, whereas OPT displays very poor performance from a model size of 6.7B (larger models with INT8 activation have even worse PPL). This could again be attributed to the Layer Norm issue within the OPT-family<sup>1</sup>.

**Findings 1 on Sensitivity Analysis.** (1) INT8 weight-only quantization can serve as a standard method for reducing memory costs in LLMs, with negligible degradation in accuracy. (2) INT4 weight-only quantization for small models results in substantial accuracy degradation (*Class-3*), but this effect lessens as the model size increases (*Class-2*). (3) Contrary to (2), INT8 activation results in minimal accuracy drops for small models (*Class-1*) but larger models exhibit greater drops (*Class-3*). (4) With INT8 activation, BLOOM shows no divergence issues up to a model size of 176B, whereas OPT performs poorly from  $\geq 6.7\text{B}$  model sizes.

## 4 Are existing quantization methods optimally harnessing the potential to minimize LLMs sizes?

Numerous lightweight optimization-based methods have been proposed, which update the model weights during quantization. These methods such as [36, 12, 35], unlike quantization-aware training, only require a small portion of the training data and a limited training time. Particularly, GPTQ [12] and ZeroQuant [36], have proven to be effective and efficient in terms of GPU resources, time cost, and data usage for INT4 weight quantization.<sup>2</sup> In this work, we focus on the variants of GPTQ and ZeroQuant as well as the most straightforward baseline, round-to-nearest neighborhood (RTN).

**RTN** directly applies PTQ on the trained data and follows the procedure detailed in Section A to perform the quantization. Specifically, for symmetric quantization, we set  $S = \max(\text{abs}(x))$  and  $Z = 0$ ; for asymmetric quantization, we set  $S = \max(x) - \min(x)$  and  $Z = \min(x)$ .

**GPTQ** extends the OBQ [10]. It tries to optimize the following non-linear least square problem,  $\min_{\hat{W}} \|Wx - \hat{W}x\|_2^2$  where  $W$  is the weight,  $x$  is the activation, and  $\hat{W}$  is a quantized weight. GPTQ employs second-order methods to obtain a closed-form solution. In addition, the quantization for each weight matrix is performed column-/row-wisely and the quantization errors from previous columns will be passed to those columns not yet quantized. See [10, 12] for more details.

**ZQ-Global** is the original method proposed in [36], where authors treat each layer as a small neural network (a.k.a., subnetwork) and use the FP16 subnetwork as the teacher model to distill the quantized one with a few hundred iterations, i.e.,  $\min_{\hat{\theta}} |f_{\theta}(x) - f_{\hat{\theta}}(x)|^2$ , where  $\theta$  is a set of weights,  $\hat{\theta}$  is the quantized version,  $f_{\theta}$  is the subnetwork with parameters  $\theta$ , and  $x$  is the input. Thus, it can significantly reduce the GPU resource requirement and time cost.

**ZQ-Local** is an extension mode of ZQ-Global for further GPU requirement reduction and training cost reduction. Particularly, instead of using each transformer layer as the subnetwork, we treat each linear layer as the subnetwork. This method can be viewed as an iterative first-order optimization method (e.g., SGD) to solve  $\min_{\hat{W}} \|Wx - \hat{W}x\|_2^2$ .

**Experimental Setup.** We compare the four methods mentioned above on weight-only and weight-and-activation quantization. As weight quantization is always static (i.e., it does not change during inference), there is virtually no system performance difference between symmetric and asymmetric quantization.<sup>3</sup> We use asymmetric quantization for better accuracy, and the conclusions would hold similarly for symmetric quantization. For parameters used for GPTQ, ZQ-Local, and ZQ-Global, please refer to Appendix B. An interesting finding for ZeroQuant is that the hyperparameters (e.g., learning rate and its scheduler) provided

<sup>2</sup>We tested the method proposed by [35] but did not find it better than others for INT4 weight quantization.

<sup>3</sup>The bias term (a.k.a., the zero point) can be simply fused into the previous activation quantization kernel [36].

Table 3: The evaluation results of different PTQ methods on OPT and BLOOM (BLM) with asymmetric quantization on weight or (and) activation. See more details in Table E.3 and Table E.6.

Precision	Method	OPT-6.7b	OPT-13b	OPT-30b	OPT-66b	BLM-1.7b	BLM-3b	BLM-7.1b	BLM-176b
W16A16		11.90	11.22	10.70	10.33	20.43	17.58	14.96	10.90
W4A16	RTN	13.44	12.09	11.52	31.52	22.47	19.01	15.90	11.20
	GPTQ	12.28	11.42	10.78	10.52	21.58	18.33	15.50	11.02
	ZQ-Local*	12.46	11.64	11.05	10.79	21.70	18.50	15.55	11.11
	ZQ-Global*	12.38	11.62	11.04	10.68	21.38	18.33	15.52	11.05
W4A8	RTN	14.80	26.36	86.26	815.00	22.75	19.17	16.19	12.22
	GPTQ	13.88	17.28	20.71	648.69	21.71	18.44	15.75	11.86
	ZQ-Local*	13.24	14.23	18.53	16.32	21.86	18.66	15.75	11.19
	ZQ-Global*	13.17	13.07	14.65	37.82	21.43	18.39	15.58	11.49

in the original work [36] are sub-optimal. In this work, we find the best configurations for ZQ-Local and ZQ-Global and denote them as ZQ-Local\* and ZQ-Global\*, respectively, with the best tuned results. To ensure consistent and comparable results, we set a fixed random seed for our experiments. In the context of post-training quantization, varying the random seed has minimal impact on the final results, as indicated in more detail in Table B.1.

**Evaluation of Weight-only Quantization.** The results from weight-only quantization using OPT and Bloom are presented in Table 3. The findings indicate that the larger models tend to be less sensitive to INT4 weight-only quantization. This observation holds true across all methods (RTN, GPTQ, ZQ-Local\*, and ZQ-Global\*) with the exception of OPT-66B, which shows greater degradation than OPT-30B. It is noteworthy that light-weight optimization-based methods significantly outperform the RTN baseline in terms of accuracy. For instance, these methods substantially reduce the degradation in perplexity of OPT-30B/66B compared to baseline. Most quantized models with parameters greater than 6.7B fall under Class II, indicating their potential for real-world applications. For instance, the quality of INT4 OPT-30B (66B) is superior to that of INT8 OPT-13B (30B).

Among the optimization-based methods, ZQ-Global\* generally performs better on smaller models (those with fewer than 1B parameters), while GPTQ excels on larger models. ZQ-Local\* does not outperform GPTQ or ZQ-Global\*—a reasonable outcome given that GPTQ employs a closed-form solution to solve the non-linear quadratic problem and ZQ-Global\* optimizes a larger subnetwork. The inferior performance of ZQ-Global\* compared to GPTQ for larger models is unexpected since ZQ-Global\* optimizes an entire transformer layer while GPTQ only optimizes a single linear layer. A plausible explanation is that larger models are more sensitive to weight updates, necessitating more advanced fine-tuning methods.

**Evaluation of Weight and Activation Quantization.** The evaluation results for existing methods using W4A8 quantization are presented in Table 3. The three light-weight optimization-based methods outperform RTN significantly, underscoring their efficacy. However, all of the results fall into either **Class-2** or **Class-3**. This suggests that for certain applications, it might be more beneficial to use smaller models with fewer parameters rather than larger, quantized models.

Among quantization-based methods, ZQ-Global\* and ZQ-Local\* generally outperform GPTQ, which is anticipated given that GPTQ was originally designed for weight-only quantization. ZQ-Global\* performs better than ZQ-Local\* in most cases except for the two largest models, OPT-66B and Bloom-176B, despite having larger trainable parameters in one step. This again signifies the need for a more suitable and advanced optimization method for large language models (LLMs).

**Finding 2 on Comparisons.** (1) GPTQ typically performs better for weight-only quantization, while ZeroQuant (including both ZQ-Global\* and ZQ-Local\*) yields superior results for weight and activation quantization. (2) The tested optimization-based methods cannot achieve **Class-1** quantization error for either INT4 weight-only or W4A8 quantization with the exception of GPTQ on OPT-30B with weight-only quantization.

Table 4: Results of **W4<sup>asym</sup>-A16** quantization with various block-size out of the best result from optimization-based methods on OPT and BLOOM (BLM). See Table E.15 and Table E.16 for full results including RTN. N/A means that the block size is not divisible by the hidden size.

Block-size	OPT-6.7b	OPT-13b	OPT-30b	OPT-66b	BLM-1.7b	BLM-3b	BLM-7.1b	BLM-176b
W16A16	11.90	11.22	10.70	10.33	20.43	17.58	14.96	10.90
Per-row	12.28	11.42	10.78	10.52	21.38	18.33	15.50	11.02
1024	12.16	11.36	10.75	10.52	31.03	N/A	15.24	10.96
512	12.08	11.32	10.73	10.52	20.93	17.99	15.20	10.95
256	12.05	11.28	10.74	10.50	20.95	17.97	15.18	10.95
128	12.10	11.28	10.74	10.44	20.92	17.90	15.17	10.94
32	12.03	11.28	10.72	10.41	20.82	17.88	15.16	10.95

#### 4.1 Fine-grained Quantization and Its Evaluation

With PTQ and row-wise quantization, achieving *Class-1* quantization error is challenging for both weight-only and weight-and-activation quantization. Generally, utilizing a smaller model with INT8 weight is more advantageous than employing a model that is twice as large with INT4 weight.

One potential solution to this issue is the implementation of finer-grained quantization schemes [5], where every  $k$  elements possess their own scaling factor and/or zero point. This approach can significantly reduce quantization error. In the extreme case, where every single element has its own scaling factor, the original FP16 number can be precisely recovered. Importantly, block- $k$  quantization can be implemented on modern GPUs, one of the most prevalent deep learning architectures, since the compute unit (streaming multiprocessor) of GPUs processes tiles of data (e.g., 128 by 128 tiling size) for matrix computation.

Although fine-grained quantization can substantially narrow the gap between the quantized tensor and its floating-point counterpart, the application of RTN still results in a non-trivial accuracy gap. Consequently, we build upon fine-grained quantization by employing existing optimization-based methods to further enhance accuracy. Specifically, we utilize GPTQ and ZQ-Global for all models and settings and apply ZQ-Local to OPT-66B and Bloom-176B. For the hyperparameters used in ZQ-Global and ZQ-Local, we select the top three identified in Section 4 for all models, except for Bloom-176B, for which we only use the top-performing hyperparameter to reduce training costs.

**4-bit Weight Quantization.** We hereby present the W4A16 results for OPT and BLOOM, as delineated in Table 4, corresponding to an array of quantization block sizes. The performance sees a significant improvement with smaller block sizes compared to per-row quantization. The point of diminishing returns, however, varies for different model sizes. For example, smaller models (such as OPT-6.7B and BLOOM-1.7b) continue to see substantial gains until the block size reduces to 32. In contrast, for larger models (those exceeding 10B, with OPT-66B as the exception), the benefits derived from smaller block sizes wane rapidly around block-256/512. Most crucially, for models equal to or larger than 13B, a smaller quantization block size results in quantization error being classified under *Class-1*, indicating virtually negligible degradation in accuracy.

Table 5: OPT W4<sup>asym</sup>-A8 with various block-size out of the best result from GPTQ, ZQ-Local, and ZQ-Global on OPT and BLOOM (BLM). See Table E.20 for full results including RTN.

Precision	block-size (W A)	OPT-6.7b	OPT-13b	OPT-30b	OPT-66b	BLM-1.7b	BLM-3b	BLM-7.1b	BLM-176b
W4A16	128   NA	12.10	11.28	10.74	10.44	20.92	17.90	15.17	10.94
W4A8	Case-1: per-row   per-row	13.17	13.07	14.65	16.32	21.43	18.39	15.58	11.19
	Case-2: per-row   128	12.29	11.45	10.80	10.61	21.59	18.31	15.52	11.03
	Case-3: 128   128	12.04	11.31	10.75	10.45	21.27	17.86	15.19	10.96

**Activation Quantization (W4A8).** To comprehend the benefits of fine-grained quantization on activation, we analyze the quantization between per-row and a block size of 128, with INT4 weight, as highlighted in Table 5. For models of considerable size, specifically those equal to or exceeding 1B, the application of such fine-grained activation quantization (Case-1) results in a substantial reduction in quantization error compared to per-row activation (Case-2). By implementing fine-grained activation quantization with weight quantization (Case-3), we are able to almost restore the performance to the level of their W4A16 counterparts.

Furthermore, we detail the impacts of varying activation quantization block sizes in Table 6 on BLOOM-176B, with INT4 weight. A trend of superior accuracy is observed with smaller block sizes in contrast to larger ones. However, the enhancement in performance reaches a saturation point when the size smaller or equal to 256, which corresponds to the range of values INT8 can represent. Despite INT8’s capability to signify 256 distinct values, activation quantization errors persist due to the application of uniform quantization.

**Finding 3 on FGQ.** (1) Larger models ( $\geq 10B$ ) are capable of attaining *Class-1* error for 4-bit quantization. These models can leverage low-precision quantization as the model size with INT4 is similar to an INT8 model that is half its size, with improved accuracy. On the other hand, smaller models ( $\leq 10B$ ) typically reach only *Class-2* or *Class-3* error levels. (2) For larger models ( $> 10B$ ), the difference between fine-grained weight-and-activation quantization and fine-grained weight-only quantization is insignificant. (3) The advantage of fine-grained activation quantization fades for larger models when the block size reaches 256.

## 5 Proposed Method to Further Push the Limit of Post-training Quantization

Building on the investigation and conclusions drawn from previous sections, it has become apparent that there is still a need for an advanced methodology to further refine the existing methods, with the objective of fully realizing the original fp16 PPL quality. In this section, we introduce a simple yet effective method called **LoRC** (Low Rank Compensation) to optimize the current existing quantization error and further bridge the gap between the quality of the original model and its quantized counterparts.

LoRC is inspired by the employment of low-rank matrix factorization on the quantization error matrix  $E := W - \hat{W}$ , where  $W$  represents the original weight and  $\hat{W}$  is the quantized weight. LoRC approximates the error  $E$  with  $\hat{E} = \hat{U}\hat{V}$  by using two low-rank matrices  $\hat{U}$  and  $\hat{V}$ . This results in a more accurate approximation of the original weight matrix  $W$  by  $\hat{W}_{\text{lorc}} = \hat{W} + \hat{E}$ , thereby reducing quantization errors:  $\|W - \hat{W}\| \geq \|W - \hat{W}_{\text{lorc}}\|$ . LoRC consists of two steps:

**Step I:** Implement Singular Value Decomposition (SVD) on the error matrix  $E = U\Sigma V$ , where  $U \in \mathbb{R}^{d_{\text{in}} \times d_{\text{in}}}$  and  $V \in \mathbb{R}^{d_{\text{out}} \times d_{\text{out}}}$  are unitary matrices, and  $\Sigma \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$  is a diagonal matrix with its diagonal elements ordered in a descending manner.

**Step II:** We formulate the matrix  $\hat{E} = \hat{U}\hat{V}$  where  $\hat{U} = U_m(\Sigma_m)^{\frac{1}{2}}$  and  $\hat{V} = (\Sigma_m)^{\frac{1}{2}}V_m$ . Here,  $U_m = U_{:,1:m} \in \mathbb{R}^{d_{\text{in}} \times m}$ ,  $V_m = V_{1:m,:} \in \mathbb{R}^{m \times d_{\text{out}}}$ , and  $\Sigma_m = \Sigma_{1:m,1:m} \in \mathbb{R}^{m \times m}$ .

The objective of LoRC is to achieve a good approximation of the error matrix  $E$  using low-rank matrices, with minimal impact on the increase in model size. For instance, consider the standard transformer models [32], where each layer is comprised of a multi-headed attention (MHA) module and a multi-linear perception (MLP) module. Let  $h$  represent the hidden dimension and  $l$  the number of layers. The total number of parameters is  $12lh^2$  as each layer contains  $4h^2$  for MHA (for key, query, value, and projection matrices), and  $8h^2$  for MLP (two matrices of sizes  $h \times 4h$  and  $4h \times h$ ). With the addition of low-rank LoRC to the six

Table 6: BLOOM-176B with different quantization block sizes on activation. Here weight is asymmetrically quantized with block size 128. See more in Table E.22.

A8 Block Size	1024	512	256	128	32
PPL	10.98	10.97	10.95	10.95	10.95

Table 7:  $W\#^{\text{asym}}$ -A16 quantization with  $\#$  being 4-bit, 3-bit and 2-bit on OPT and BLOOM (BLM).

Bits	LoRC	Coarse-grained weight quantization (per-row block-size)					Fine-grained quantization on weight (256 block-size)				
		OPT-6.7b	OPT-13b	OPT-30b	OPT-66b	BLM-176b	OPT-6.7b	OPT-13b	OPT-30b	OPT-66b	BLM-176b
W8A16		11.90	11.22	10.70	10.33	10.90	11.90	11.22	10.70	10.33	10.90
W4A16	✗	12.28	11.42	10.78	10.78	11.02	12.05	11.28	10.74	10.50	10.95
	✓	12.10	11.36	10.76	10.34	10.98	11.99	11.29	10.70	10.29	10.93
W3A16	✗	14.18	12.43	11.28	17.77	49.46	12.79	11.63	10.9	11.34	11.13
	✓	13.00	11.90	11.14	10.63	11.30	12.40	11.57	10.83	10.42	11.08
W2A16	✗	120.56	40.17	25.74	225.45	Explode	23.13	15.55	12.68	308.49	12.64
	✓	24.17	18.53	14.39	13.01	14.15	16.27	14.30	12.37	11.54	12.21

matrices in each layer, the total number of parameters for  $l$  layers would amount to  $18hml$ .<sup>4</sup> Consequently, the ratio of parameters added to the existing model is  $3m/2h$ . It's important to note that the low-rank dimension  $m$  can be as small as 4 or 8 (which we will discuss in detail in a later section) while the standard hidden dimension  $h \geq 768$ , making the number  $3m/2h \leq 0.016$ .

Significantly, LoRC can be viewed as a supplementary feature to existing quantization methodologies such as RTN, GPTQ, and ZeroQuant-Local/Global, and can be seamlessly integrated with FGQ. We have conducted experiments to evaluate the performance of LoRC on both OPT and BLOOM, applying 4-bit, 3-bit, and 2-bit weights by setting the activation to FP16.<sup>5</sup> Based on the discoveries in the preceding sections, we utilize the GPTQ quantization strategy. To gain a comprehensive understanding of LoRC, we include the results with and without the application of FGQ. The datasets and hyperparameters are consistent with those detailed in earlier sections.

**Evaluation Results.** The findings are showcased in Table 7, split into two sections: coarse-grained weight quantization (per-row) and fine-grained quantization (block-size 256). Notably, we observe that the two low-rank matrices,  $\hat{U}$  and  $\hat{V}$ , can be quantized to 8-bit without any performance discrepancy (Table 8). Thus, the two low-rank matrices for LoRC in Table 7 are INT8 with a low-rank dimension of  $m = 8$ .

Several key observations can be made. Firstly, LoRC consistently boosts performance across all bit sizes and block sizes, as indicated by the lower perplexity scores when LoRC is activated. Secondly, the enhancement brought about by LoRC becomes more substantial as the bit size diminishes, especially noticeable for W2A16, which displays a markedly greater impact compared to W4A16 and W3A16 in most scenarios. Lastly, the combination of fine-grained quantization with LoRC yields the most impressive results, underscoring the efficacy of LoRC when integrated with FGQ. Overall, the results emphasize the benefits of using LoRC for enhanced performance in weight quantization and its compatibility with FGQ. Notably, recovering the last 0.05-0.1 perplexity can be challenging, but with LoRC, we are able to nearly recover the original model quality for INT4 quantization.

**Ablation Study on the Low Rank Dimension  $m$ .** An essential aspect of the LoRC method is on the optimal low-rank dimension, denoted as  $m$ , explained in **Step II**. To explore this, we varied  $m$  in the range of 1, 4, 8, 16, and 32 for OPT-1.3b/6.7b/30b models, and applied W4A16 GPTQ quantization. The outcomes are depicted in Table 9, indicating that the enhancements achieved through LoRC begin to plateau as the dimension  $m$  surpasses 4. The most optimal performance for OPT-6.7b is realized when  $m = 8$ .

This observation may seem counterintuitive initially, as one might anticipate that larger LoRC dimensions

<sup>4</sup>In the MHA module, LoRC contributes  $2hm$  to each of key, query, value, and the projection matrices. In the MLP module, LoRC contributes  $8hm$  and  $2hm$  respectively to the matrices of dimensions  $h \times 4h$  and  $4h \times h$ .

<sup>5</sup>For INT8 Activation, please see Table E.23, the observation for FP16 holds similarly for INT8 Activation.

Table 9: W4A16 quantization with LoRC by varying the low-rank dimension  $m$ .

LoRC-dim $m$	OPT-1.3b	OPT-6.7b	OPT-30b
$m = 0$ baseline	15.95	12.06	10.73
$m = 1$	15.93	12.01	10.73
$m = 4$	15.73	12.00	10.72
$m = 8$	15.76	11.99	10.70
$m = 16$	15.74	12.00	10.69
$m = 32$	15.71	12.01	10.69

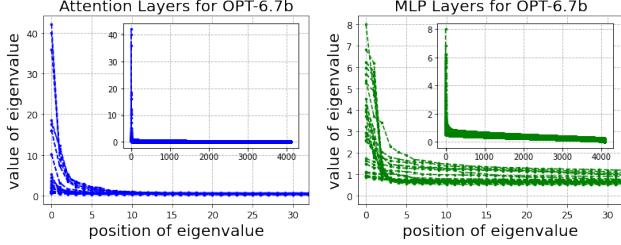


Figure 2: Eigenvalues of the Error matrix  $E$  for W4A16

would yield more significant improvements. To gain a more comprehensive understanding, we conducted an analysis of the eigenvalues of the actual error matrix  $E = W - \hat{W}$  for each matrix. By randomly selecting 20 matrices from MHA and MLP layers, we plotted the eigenvalues of  $E$  as a curve, depicted in Figure 2. The two plots reveal a rapid flattening of eigenvalues after index 8, which elucidates why increasing the LoRC dimension does not considerably enhance performance. Hence, a sensible dimension for  $\hat{U}$  and  $\hat{V}$  in the LoRC methodology could be 8.<sup>6</sup>

## 6 Discussion

**Conclusion.** In this work, we provide a comprehensive study of post-training quantization (PTQ) on large language models with different PTQ methods (e.g., RTN, GPTQ, ZeroQuant), and with different quantization coverage (weight-only and weight-and-activation quantization), etc. We find that PTQ methods are critical to improving the quantized model quality, and that fine-grained quantization (FGQ) can bring acceptable accuracy and model size trade-off. Finally, we introduced an optimization technique called Low Rank Compensation (LoRC), which works synergistically with PTQ and FGQ, playing a crucial role in enhancing full model quality recovery with a minimal increase in model size.

**Limitation.** Despite quantizing over 10,000 experiments, our study was constrained by our computing resources. This restriction made us choose between diversifying the model sizes and varying the tasks. We strategically limited our datasets to WikiText, PTB, and C4 to concentrate on a broad range of quantization methods. Consequently, our general findings are more robust concerning the two model families and three datasets examined in this paper. However, caution should be exercised when generalizing these findings to tasks that are dissimilar to those covered in this study.

**Future Opportunity.** Throughout the paper, we see several unresolved problems from current quantization schemes and/or algorithms, and we find potential directions for LLM compression: (1) Although we use fine-grained quantization schemes in the paper, the real implementation is missing. Moreover, how to efficiently implement odd bit precision is challenging. [12] demonstrated that 3-bit can achieve better throughput in the generation phase by packing all 3-bit numbers in continuous memory space. However, this method is sub-optimal as the dequantization step needs to connect bits from different bytes. One possible way to implement odd bits, e.g., 5 bits, is to use two integer matrices with INT4 and INT1. During the dequantization stage, we couple the two matrices together. (2) How to combine PTQ with other lightweight compression techniques, e.g., post-training pruning [20, 11], is an interesting direction to further reduce the memory consumption and compute cost.

<sup>6</sup>Please note that this observation is only true for PTQ. If one uses quantize-aware training (QAT) and let  $\hat{U}$  and  $\hat{V}$  updated during QAT, we arrive at contrasting conclusions. For more details, please refer to Appendix D.

## References

- [1] Haoli Bai, Wei Zhang, Lu Hou, Lifeng Shang, Jing Jin, Xin Jiang, Qun Liu, Michael Lyu, and Irwin King. Binarybert: Pushing the limit of bert quantization. *arXiv preprint arXiv:2012.15701*, 2020.
- [2] Big-Science. Bloom inference. [https://github.com/huggingface/transformers-bloom-inference /tree/main/bloom-inference-scripts](https://github.com/huggingface/transformers-bloom-inference/tree/main/bloom-inference-scripts), 2022.
- [3] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Understanding and overcoming the challenges of efficient transformer quantization. *arXiv preprint arXiv:2109.12948*, 2021.
- [4] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.
- [5] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, et al. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. *Advances in neural information processing systems*, 33:10271–10281, 2020.
- [6] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.
- [7] Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. *arXiv preprint arXiv:2212.09720*, 2022.
- [8] Steven K Esser, Jeffrey L McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S Modha. Learned step size quantization. *arXiv preprint arXiv:1902.08153*, 2019.
- [9] Angela Fan, Pierre Stock, Benjamin Graham, Edouard Grave, Remi Gribonval, Herve Jegou, and Armand Joulin. Training with quantization noise for extreme fixed-point compression. *arXiv preprint arXiv:2004.07320*, 2020.
- [10] Elias Frantar and Dan Alistarh. Optimal brain compression: A framework for accurate post-training quantization and pruning. *arXiv preprint arXiv:2208.11580*, 2022.
- [11] Elias Frantar and Dan Alistarh. Massive language models can be accurately pruned in one-shot. *arXiv preprint arXiv:2301.00774*, 2023.
- [12] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.
- [13] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.
- [14] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W Mahoney, and Kurt Keutzer. Ai and memory wall. *RiseLab Medium Post*, 2021.
- [15] GitHub. Github copilot. <https://github.com/features/copilot/>, 2021.
- [16] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171, 1993.
- [17] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *Workshop paper in NIPS*, 2014.
- [18] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351*, 2019.

- [19] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. I-bert: Integer-only bert quantization. In *International conference on machine learning*, pages 5506–5518. PMLR, 2021.
- [20] Woosuk Kwon, Sehoon Kim, Michael W Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. A fast post-training pruning framework for transformers. *arXiv preprint arXiv:2204.09656*, 2022.
- [21] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.
- [22] Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Using Large Corpora*, page 273, 1994.
- [23] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017.
- [24] OpenAI. Openai chatgpt. <https://openai.com/blog/chatgpt/>, 2022.
- [25] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018.
- [26] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*, 2022.
- [27] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019.
- [28] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.
- [29] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-BERT: Hessian based ultra low precision quantization of bert. In *AAAI*, pages 8815–8821, 2020.
- [30] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.
- [31] Chaofan Tao, Lu Hou, Wei Zhang, Lifeng Shang, Xin Jiang, Qun Liu, Ping Luo, and Ngai Wong. Compression of generative pre-trained language models via quantization. *arXiv preprint arXiv:2203.10705*, 2022.
- [32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
- [33] Xiaoxia Wu, Cheng Li, Reza Yazdani Aminabadi, Zhewei Yao, and Yuxiong He. Understanding int4 quantization for transformer models: Latency speedup, composability, and failure cases. *arXiv preprint arXiv:2301.12017*, 2023.
- [34] Xiaoxia Wu, Zhewei Yao, Minjia Zhang, Conglong Li, and Yuxiong He. Extreme compression for pre-trained transformers made simple and efficient. *arXiv preprint arXiv:2206.01859*, 2022.
- [35] Guangxuan Xiao, Ji Lin, Mickael Seznec, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*, 2022.

- [36] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *arXiv preprint arXiv:2206.01861*, 2022.
- [37] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 811–824. IEEE, 2020.
- [38] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8BERT: Quantized 8bit bert. *arXiv preprint arXiv:1910.06188*, 2019.
- [39] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414*, 2022.
- [40] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuhui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.
- [41] Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. Ternarybert: Distillation-aware ultra-low bit bert. *arXiv preprint arXiv:2009.12812*, 2020.

## A Background of Quantization

Quantization maps floating point (e.g., FP16/FP32) numbers to integer numbers (e.g., INT4/INT8) so that lower memory usage (weight quantization) and faster integer arithmetic (weight-and-activation quantization) can be achieved compared to the floating point format. In this work, we are focusing on uniform quantization, i.e.,

$$Q(x) = \text{INT}((x - Z)/S) - Z, \quad (1)$$

where  $Q$  is the quantization function,  $x$  is a floating point input vector/tensor,  $S$  is a real valued scaling factor, and  $Z$  is an integer zero point. Based on different settings, the quantization method can be viewed as (1) symmetric vs. asymmetric quantization ( $Z = 0$  or not), (2) fine-grained vs. coarse-grained quantization (how to partition the input  $x$  and get its associated scaling factor, e.g., matrix wise or row wise). See [13] for more details.

Throughout this work, we focus on post-training quantization (PTQ), i.e., no or minimal training effort is applied after quantization, for which large accuracy degradation usually exhibits for coarse-grained quantization (per matrix/tensor) due to their large quantization error. As such, we focus on fine-grained quantization. Particularly, we use the per-row quantization (one row of the weight matrix or one token for the activation) from [36] as our coarsest-grained quantization method, and we use block-k quantization (for every  $k$  elements, they have their own scaling factor and/or zero point) as our finer-grained quantization scheme.

## B Detailed Setting Used in Section 4

Same as [12], for all methods, we use C4 dataset to randomly select 128 sentences for training and each of them has 2048 tokens.

For GPTQ, we check its main hyperparameter, i.e., the dampening factor, and find out the method is not sensitive to it. As such, we use the hyperparameter suggested by the author for all of our experiments. For ZQ-Global and ZQ-Local, as mentioned in the main text, the hyperparameters suggested by the original work [36] is suboptimal. We find that a linear decay learning rate schedule is very helpful in our initial test. As such, we add this as our default setting. Meanwhile, we extensively test a wide range (1e-3 to 5e-8) of learning rate for different models until we find the best learning rate (i.e., larger or smaller learning rate leads to worse accuracy performance). We employed the Adam optimizer and set the default batch size to 1 for our experiments.

We conducted tests to assess whether changes in random seeds would introduce substantial variations in the outcomes. As per the findings detailed in Table Table B.1, the modifications in random seeds resulted in only minimal effects on the final quality of the models. This effect was particularly negligible in the context of larger models, such as OPT-30b, where the standard deviation was only 0.01. Therefore, in consideration of these results, we elected to standardize the random seed for the subsequent experiments presented in this paper, setting it uniformly at 123 or 0. The code will be made publicly available to facilitate reproducibility of our results.

For all three methods, we run them on a single GPU (either V100-32GB or A100-80GB). For the largest model tested in the paper, i.e., BLOOM-176B, the cost of all methods is lower than one GPU-day on A100-80G.

Table C.1: Best optimization method of OPT family in Section 4.

Precision	125m	350m	1.3b	2.7b	6.7b	13b	30b	66b
Weight Only (INT4)	ZQ-Global	ZQ-Global	GPTQ	GPTQ	GPTQ	GPTQ	GPTQ	GPTQ
Weight & Activation (W4A8)	ZQ-Global	ZQ-Global	ZQ-Global	GPTQ	ZQ-Global	ZQ-Global	ZQ-Global	ZQ-Local

Table C.2: Best optimization method of BLOOM family in Section 4.

Precision	560m	1.1b	1.7b	3b	7.1b	176b
Weight Only (INT4)	GPTQ	ZQ-Global	ZQ-Global	ZQ-Global/GPTQ	GPTQ	GPTQ
Weight & Activation (W4A8)	ZQ-Global	ZQ-Global	ZQ-Global	ZQ-Global	ZQ-Global	ZQ-Local

Table B.1: The table on the left illustrates the outcomes of each task, evaluated using three different random seeds. On the right, we present a table detailing the mean and standard deviation of the Task-mean values (which can be found in the final column of the left table) over the three random seeds, accompanied by additional quantization results. The quantization methodologies employed in this context are based on the GPTQ algorithm.

Precision	Random Seed	WikiText	PTB	C4	Task-mean	Precision	Items	OPT-1.3b	OPT-13b	OPT-30b
OPT-13b W4A16	123	10.31	12.62	11.35	11.43	W4A16	mean over three random seeds	16.39	11.42	10.77
	234	10.25	12.57	11.35	11.39		standard deviation	0.019	0.027	0.010
	456	10.37	12.61	11.36	11.44		mean over three random seeds	16.76	17.16	21.64
OPT-30b W4A16	123	9.56	11.95	10.79	10.77		standard deviation	0.048	0.048	1.277
	234	9.6	11.95	10.79	10.78					
	456	9.52	11.97	10.79	10.76					

## C Best PTQ Methods with Per-row Quantization

Table C.1 and C.2 summarize the best PTQ methods with per-row optimization.

## D Quantization-aware training with LoRC

In order to better understand our proposed algorithm, LoRC, particularly in relation to the dimensions of low-rank matrices, we applied quantize-aware training alongside knowledge distillation. This approach builds upon the methodology of row-wise weight quantization and token-wise quantization. For the optimization process, we employed the Adam optimizer, setting the learning rate at 1e-4 and a dropout rate of 0.05. These settings were identified as the most effective in our context (additional details can be found in [33]). We performed fine-tuning on the WikiText dataset using pre-trained GPT2 models with 125M and 350M parameters, which were obtained from Hugging Face as our initial models.<sup>7</sup>

The results are illustrated in Figure Figure D.1. As observed, the quantized models tend to overfit swiftly. However, implementing higher dropout values, such as 0.1, does not result in a significantly improved performance with regards to the best perplexity over the entire training duration. Now when examining the best perplexity associated with each dimension of LoRC (also indicated in the figure's legend), it becomes evident that the larger the dimension, the better the W4A8 models perform. This suggests that augmenting the dimension of LoRC can enhance the model quality for QAT, a finding that deviates from the trends observed in PTQ.

---

<sup>7</sup><https://huggingface.co/gpt2>

Table E.1: OPT ppl on wikitext/ptb/c4 (full results of Table 2).

Precision	125m	350m	1.3b	2.7b	6.7b	13b	30b	66b
W16-A16	27.65/32.55/24.61	22.00/26.08/20.71	14.62/16.97/14.72	12.47/15.11/13.17	10.86/13.09/11.74	10.13/12.34/11.20	9.56/11.84/10.69	9.34/11.36/10.28
W8Asym-A16	27.64/32.53/24.65	22.06/26.10/20.72	14.63/16.98/14.73	12.48/15.13/13.17	10.85/13.11/11.75	10.12/12.34/11.20	9.55/11.85/10.70	9.34/11.36/10.29
W8asymp-A16	27.71/32.58/24.64	22.04/26.12/20.73	14.67/16.99/14.73	12.50/15.14/13.17	10.86/13.11/11.75	10.11/12.34/11.20	9.55/11.84/10.69	9.35/11.36/10.29
W4sym-A16	45.89/53.68/36.68	25.95/31.11/23.94	19.85/23.61/18.90	22.86/30.01/22.29	12.41/17.05/13.62	11.06/14.90/12.23	10.18/13.26/11.86	57.73/134.91/98.51
W4asymp-A16	36.71/44.76/30.92	25.51/30.90/23.86	19.38/21.95/17.93	17.92/22.48/18.32	11.91/15.39/13.01	10.67/13.53/12.07	10.10/13.13/11.33	20.24/48.45/25.86
W16-A8sym	27.96/32.57/24.69	22.06/26.42/20.95	15.21/18.18/15.81	12.98/16.01/13.89	20.99/25.94/31.18	3341.50/2618.38/3554.59	1681.48/2221.62/2241.53	2696.91/2647.41/2569.94
W16-A8asymp	27.84/32.60/24.66	22.04/26.22/20.81	15.14/17.65/15.39	12.51/15.38/13.38	11.24/14.17/12.45	11.83/18.87/15.39	14.08/31.54/25.09	442.66/524.57/716.83

Table E.2: BLOOM ppl on wikitext/ptb/c4 (full results of Table 2).

Precision	560m	1.1b	1.7b	3b	7.1b	176b
W16-A16	22.43/41.25/24.38	17.69/46.98/20.29	15.39/27.93/17.97	13.48/23.12/16.14	11.37/19.40/14.13	8.11/13.62/10.97
W8sym-A16	22.44/41.28/24.39	17.70/47.01/20.29	15.40/27.91/17.98	13.49/23.14/16.14	11.37/19.40/14.13	8.11/13.63/10.98
W8asymp-A16	22.43/41.24/24.40	17.69/47.00/20.29	15.40/27.96/17.97	13.48/23.14/16.14	11.37/19.40/14.13	8.10/13.62/10.98
W4sym-A16	26.49/49.73/27.98	20.27/56.64/22.81	17.47/32.20/19.88	14.96/25.59/17.51	12.38/21.36/15.06	8.40/14.15/11.30
W4asymp-A16	25.31/46.79/27.10	23.90/68.31/25.99	16.93/31.02/19.47	14.65/25.12/17.26	12.06/20.83/14.83	8.34/14.03/11.23
W16-A8sym	22.50/41.58/24.46	17.78/47.28/20.38	15.57/28.36/18.13	13.57/23.38/16.25	11.58/19.92/14.35	8.75/14.94/12.61
W16-A8asymp	22.45/41.37/24.42	17.71/47.05/20.32	15.45/28.09/18.02	13.52/23.24/16.19	11.47/19.71/14.25	8.41/14.52/11.93

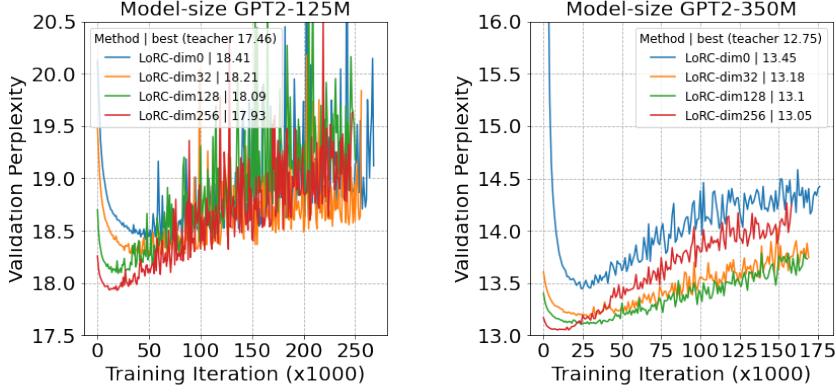


Figure D.1: The graph on the left represents the results for a smaller model size (GPT2-125M), while the one on the right corresponds to the GPT2-350M model. The dimension (refer to the legend) in the LoRC algorithm, which is represented by different color curves, plays a pivotal role in approximating the original quality of the fp16 model.

## E Tables and Figures

We put the full results of our evaluations in this section.

Table E.3: OPT ppl on wikitext/opt/c4 with W4<sup>asym</sup>-A16 (full table of Table 3). See Table E.4 for all learning rate results of ZQ-Local and Table E.5 of ZQ-Global.

Precision	125m	350m	1.3b	2.7b	6.7b	13b	30b	66b
RTN	36.71/44.76/30.92	25.51/30.90/23.86	19.38/21.95/17.93	17.92/22.48/18.32	11.91/15.39/13.01	10.67/13.53/12.07	10.10/13.13/11.33	20.24/48.45/25.86
GPTQ	32.52/40.25/27.78	23.50/29.14/22.41	15.52/18.16/15.56	13.02/15.84/13.73	11.16/13.59/12.08	10.29/12.61/11.35	9.61/11.95/10.79	9.54/11.67/10.52
ZQ-Local*	33.05/39.34/28.11	24.40/29.22/22.82	15.81/18.66/15.76	13.22/16.19/13.96	11.32/13.79/12.26	10.42/12.90/11.60	9.97/12.32/11.03	9.91/11.87/10.59
ZQ-Global*	31.44/36.66/27.21	23.32/28.05/21.98	15.46/18.31/15.67	13.03/16.04/13.83	11.30/13.69/12.17	10.38/12.85/11.62	9.90/12.24/10.99	9.62/11.81/10.61

Table E.4: OPT ppl on wikitext/opt/c4 with W4<sup>asym</sup>-A16 and ZQ-Local.

LR (W4 <sup>asym</sup> -A16)	125m	350m	1.3b	2.7b	6.7b	13b	30b	66b
0.001	33.67/39.45/29.11	26.33/31.94/24.49	16.27/19.91/16.46	14.34/17.76/14.93	11.87/15.04/13.06	13.68/18.89/14.46	171.35/151.55/46.14	814.22/601.74/308.53
0.0005	32.76/39.51/28.64	25.88/30.95/23.96	16.29/19.82/16.27	14.16/17.65/14.79	11.92/15.23/12.95	10.93/13.82/12.03	10.23/13.46/11.44	10.10/12.27/10.81
0.0001	33.86/40.01/28.29	24.64/26.23/23.33	16.07/19.25/15.93	14.36/17.38/14.41	11.85/14.6/12.74	10.93/13.48/11.16	10.18/12.67/11.13	10.12/12.01/10.67
5e-05	33.05/39.34/28.11	25.42/29.65/23.22	15.79/19.16/15.88	13.70/16.80/14.16	11.71/14.32/12.41	10.75/13.38/11.77	9.95/12.54/11.09	10.02/11.89/10.64
1e-05	33.78/40.41/28.84	24.40/29.22/22.82	15.81/18.66/15.76	13.55/16.46/14.39	11.32/13.79/12.26	10.54/13.05/11.61	9.98/12.22/10.99	9.91/11.87/10.59
5e-06	34.47/41.04/29.02	25.40/29.27/23.00	16.01/18.73/15.91	13.22/16.19/13.96	11.33/13.86/12.29	10.42/12.90/11.60	9.86/12.33/10.97	9.97/11.86/10.60
1e-06	35.88/43.69/30.35	24.54/29.87/23.17	16.77/19.45/16.47	13.60/17.02/14.46	11.41/14.10/12.41	10.53/13.01/11.70	9.97/12.33/11.04	10.01/11.93/10.66

Table E.5: OPT ppl on wikitext/opt/c4 with W4<sup>asym</sup>-A16 and ZQ-Global. NaN here means the PPL is larger than 1e6.

LR (W4 <sup>asym</sup> -A16)	125m	350m	1.3b	2.7b	6.7b	13b	30b	66b
0.001	4057.13/2718.91/1247.78	5071.35/5229.93/687.35	12105.25/10154.73/7803.43	18965.76/17112.60/16316.31	60014.66/56941.86/78085.84	292421.09/28305.32/119762.73	93037.09/70170.34/51124.06	NaN
0.0005	31.94/38.61/27.17	27.11/33.91/24.07	10900.84/8322.65/8425.10	14412.30/8676.75/10154.55	18827.46/13530.12/13629.95	100906.53/12544.51/12549.50	30235.75/230559.62/436480.03	36439.32/30554.19/33376.93
0.0001	31.44/36.96/27.21	24.08/29.08/22.27	15.91/20.08/16.35	118.38/53.47/54.08	7604.92/5339.10/5161.49	12838.86/7639.95/8243.63	16276.68/1729.21/3743.01	8367.31/4728.13/5533.59
5e-05	31.97/36.95/27.12	23.55/28.06/22.02	15.82/18.65/15.65	13.80/16.44/13.97	26.54/25.67/17.60	909.99/316.82/370.84	6238.21/3291.04/3743.01	9296.98/6687.44/5363.29
1e-05	32.31/37.93/27.38	23.32/28.05/21.98	15.60/18.42/15.64	13.09/16.05/13.78	11.41/13.82/12.20	10.80/13.16/11.66	10.06/12.44/11.07	9.73/12.09/10.98
5e-06	32.69/37.91/27.76	23.26/28.33/22.05	15.46/18.41/15.67	13.03/16.05/13.83	11.30/13.80/12.17	10.80/13.16/11.66	9.90/12.24/10.99	9.58/12.17/10.78
1e-06	34.63/41.75/29.43	23.82/29.96/22.48	16.12/18.40/16.27	13.03/16.27/14.04	11.29/13.88/12.27	10.38/12.85/11.62	9.89/12.41/11.04	9.90/12.45/11.00
5e-07	NaN	NaN	NaN	NaN	NaN	10.51/12.96/11.70	10.63/13.29/11.89	10.02/12.82/11.18
1e-07	NaN	NaN	NaN	NaN	NaN	10.66/13.42/11.97	10.05/13.00/11.24	12.41/17.45/13.02
5e-08	NaN	NaN	NaN	NaN	NaN			

Table E.6: BLOOM ppl on wikitext/opt/c4 with W4<sup>asym</sup>-A16 (full table of Table 3). See Table E.4 for all learning rate results of ZQ-Local and Table E.5 of ZQ-Global.

Precision	560m	1.1b	1.7b	3b	7.1b	176b
RTN	25.31/46.79/27.10	23.90/68.31/25.99	16.93/31.02/19.47	14.65/25.12/17.26	12.06/20.83/14.83	8.34/14.03/11.23
GPTQ	23.90/43.76/25.59	24.34/68.10/26.58	16.36/29.58/18.79	14.10/24.23/16.66	11.80/20.23/14.47	8.22/13.78/11.07
ZQ-Local*	24.23/44.94/26.05	19.22/52.36/21.59	16.37/29.89/18.86	14.23/24.41/16.86	11.80/20.28/14.56	8.27/13.91/11.16
ZQ-Global*	23.84/44.17/25.60	19.50/51.33/21.72	16.19/29.28/18.66	14.14/24.16/16.69	11.77/20.27/14.52	8.24/13.82/11.10

Table E.7: BLOOM ppl on wikitext/opt/c4 with W4<sup>asym</sup>-A16 and ZQ-Local.

LR (W4 <sup>asym</sup> -A16)	560m	1.1b	1.7b	3b	7.1b	176b
0.001	25.37/47.36/27.03	19.89/53.86/22.11	16.70/31.19/19.30	14.45/25.28/17.16	12.22/21.34/15.04	8.82/15.77/11.98
0.0005	25.17/46.83/26.87	19.57/53.66/21.92	16.58/30.27/19.15	14.43/25.47/17.07	11.94/20.54/14.67	8.35/14.01/11.20
0.0001	24.59/46.11/26.32	19.22/52.36/21.59	16.41/30.29/18.90	14.35/24.81/16.87	11.83/20.34/14.58	8.28/13.92/11.14
5e-05	24.44/46.04/26.16	23.28/65.68/25.42	16.39/30.01/18.86	14.34/24.43/16.83	11.80/20.28/14.56	8.27/13.93/11.15
1e-05	24.23/44.94/26.05	23.45/66.29/25.52	16.37/29.89/18.86	14.23/24.41/16.86	11.84/20.39/14.58	8.27/13.91/11.16
5e-06	24.21/45.21/26.10	23.26/65.72/25.42	16.42/30.09/18.94	14.25/24.55/16.87	11.87/20.50/14.61	8.29/13.98/11.16
1e-06	24.71/45.86/26.50	23.45/66.28/25.56	16.64/30.52/19.15	14.46/24.76/17.04	11.94/20.55/14.70	8.29/13.97/11.18

Table E.8: BLOOM ppl on wikitext/opt/c4 with W4<sup>asym</sup>-A16 and ZQ-Global.

LR (W4 <sup>asym</sup> -A16)	560m	1.1b	1.7b	3b	7.1b	176b
0.001	6853935.00/30441738.00/322287.25	528072.88/828428.62/356031.97	597410.50/973155.88/1280478.12	878460.69/2175974.25/441401.94	nan/nan/nan	NaN
0.0005	29671.52/1795030.88/4653.35	28112.96/87515.64/1826.82	141110.14/204295.86/40146.11	265457.25/741326.38/99882.45	944784.19/774538.25/395960.03	NaN
0.0001	23.92/45.68/25.72	19.34/52.78/21.63	16.35/29.22/18.76	14.27/24.46/16.80	12.17/22.16/14.80	NaN
5e-05	23.84/44.17/25.60	19.50/51.33/21.72	16.19/29.28/18.66	14.14/24.16/16.69	11.81/20.41/14.50	NaN
1e-05	23.85/44.20/25.65	22.64/56.79/23.41	16.23/29.73/18.73	14.14/24.31/16.74	11.77/20.27/14.52	8.24/13.82/11.10
5e-06	24.02/44.62/25.79	23.46/63.27/24.88	16.28/29.83/18.81	14.19/24.38/16.80	11.77/20.33/14.54	8.24/13.82/11.10
1e-06	24.46/45.41/26.20	24.62/70.16/26.64	16.48/30.15/19.02	14.35/24.56/16.95	11.89/20.54/14.67	8.23/13.82/11.12
5e-07	NaN	NaN	NaN	NaN	NaN	8.26/13.86/11.13

Table E.9: OPT ppl on wikitext/opt/c4 with W4<sup>asym</sup>-A8<sup>sym</sup>/A8<sup>asym</sup>. See Table E.10 for all learning rate results of ZQ-Local and Table E.11 of ZQ-Global.

Precision	125m	350m	1.3b	2.7b	6.7b	13b	30b	66b
W4 <sup>asym</sup> -A8 <sup>sym</sup> Block								
RTN	36.69/44.34/30.60	26.59/32.13/24.81	25.31/26.89/22.01	30.84/35.73/29.01	164.51/110.85/162.94	4460.61/3145.51/4255.84	3216.45/2929.40/3570.19	3038.22/2930.92/3001.82
GPTQ	32.20/38.49/27.47	24.35/29.82/23.24	16.28/19.64/16.73	13.86/17.51/15.00	46.22/53.98/55.13	3611.71/2796.71/3820.57	1738.44/1810.08/2119.82	5992.87/4115.01/4360.16
ZQ-Local*	32.88/38.23/28.20	25.18/30.06/23.62	16.78/20.25/17.09	14.82/18.77/15.61	16.08/21.15/18.77	2680.33/1876.48/3052.51	1884.90/1603.23/1348.08	575.20/499.42/437.94
ZQ-Global*	32.04/37.48/27.23	24.01/28.81/22.57	16.12/19.15/16.23	13.98/17.70/14.87	38.27/39.77/52.26	117.83/141.63/96.83	253.71/700.40/337.15	1715.98/1546.50/1799.35
W4 <sup>asym</sup> -A8 <sup>asym</sup> Block								
RTN	36.61/44.48/30.64	25.79/31.28/24.13	21.23/23.54/19.19	23.82/29.77/22.60	13.18/17.04/14.19	19.87/32.93/26.28	36.07/136.88/85.84	627.15/880.79/937.08
GPTQ	32.22/38.83/27.43	23.90/29.29/22.63	15.75/18.74/15.93	13.23/16.31/14.03	12.50/15.86/13.29	12.79/21.99/17.05	12.96/25.03/24.14	495.70/681.68/768.69
ZQ-Local*	33.60/38.57/28.02	24.57/29.27/22.98	15.98/19.13/16.20	13.44/16.81/14.26	11.76/14.97/13.00	11.69/16.98/14.01	12.38/24.25/18.96	12.19/23.31/13.47
ZQ-Global*	31.61/37.00/27.10	23.66/28.56/22.21	15.77/18.61/15.83	13.09/16.56/14.00	12.03/14.60/12.86	11.80/15.01/12.41	12.94/17.61/13.41	31.51/58.00/23.95

Table E.10: OPT ppl on wikitext/opt/c4 with W4<sup>asym</sup>-A8<sup>sym</sup>/A8<sup>asym</sup> and ZQ-Local.

Precision	125m	350m	1.3b	2.7b	6.7b	13b	30b	66b
W4 <sup>asym</sup> -A8 <sup>sym</sup> Block								
0.001	34.91/40.43/29.37	26.82/32.68/25.24	17.68/21.72/18.11	19.40/27.59/20.05	36.70/59.32/45.17	7240.89/5506.67/4889.34	8229.32/5068.14/5005.13	Diverge
0.0005	34.16/39.00/28.58	26.75/32.05/24.60	17.19/21.42/17.55	19.43/25.54/19.41	29.33/48.38/43.28	56836.57/36810.64/31073.67	5448.96/3826.63/3196.49	575.10/499.42/437.94
0.0001	32.88/38.23/28.20	25.31/31.60/23.98	15.75/18.74/15.93	13.23/16.31/14.03	17.05/21.50/17.42	25.24/31.66/26.82	6125.07/3817.01/4122.70	1884.90/1603.23/1348.08
5e-05	32.86/39.17/27.91	25.91/31.24/24.07	16.99/20.02/17.23	15.07/19.00/15.54	16.08/21.15/18.77	6037.51/3617.64/3819.63	3266.46/2533.64/2463.21	11631.78/10489.81/7880.43
1e-05	34.00/39.76/28.62	25.40/30.60/23.75	16.87/20.26/17.11	14.82/18.77/15.61	26.60/32.09/29.57	5346.85/3788.29/4903.31	3364.70/2372.71/3370.97	5793.44/3544.90/3925.34
5e-06	34.37/41.46/28.71	25.18/30.06/23.62	16.78/20.25/17.09	14.87/19.42/15.86	34.53/39.98/38.22	2680.33/1876.48/3052.51	3566.45/2532.54/3678.75	4916.96/3783.69/3716.49
1e-06	36.05/43.46/30.00	25.73/30.69/24.05	19.58/22.57/19.04	18.66/24.19/19.98	77.99/62.27/83.19	3893.00/2672.11/3849.59	3233.72/2944.44/3732.18	4238.57/3621.09/3541.33
W4 <sup>asym</sup> -A8 <sup>asym</sup> Block								
0.001	33.57/40.86/29.00	27.29/32.48/24.68	17.41/20.70/17.07	15.98/20.45/16.23	12.63/17.21/14.25	9889.96/7605.56/6328.91	2009.66/1637.69/2011.15	5070.07/3124.56/2683.19
0.0005	34.58/40.45/28.60	25.81/31.56/24.09	16.89/20.66/16.93	15.00/19.47/15.61	12.55/17.00/14.29	13.18/19.65/15.18	36.51/75.89/60.58	3249.10/63/119.55
0.0001	33.91/38.39/28.12	25.37/31.24/23.66	16.78/20.09/16.72	14.26/18.49/14.90	12.13/15.97/13.48	13.48/20.42/16.68	110.20/117.28/257.96	12.19/23.31/13.47
5e-05	33.60/38.57/28.02	24.67/29.60/23.34	16.31/19.56/16.42	13.90/19.16/15.05	12.30/15.95/13.56	12.05/18.00/15.77	37.68/59.83/124.75	29.72/95.99/69.60
1e-05	33.80/40.21/28.56	24.57/29.27/22.98	15.98/19.13/16.20	13.44/16.81/14.26	11.76/14.97/13.00	11.69/16.98/14.01	14.39/31.47/24.45	217.93/313.13/298.24
5e-06	34.62/41.07/28.93	24.68/29.46/23.12	16.26/19.23/16.27	13.44/17.00/14.36	11.96/14.86/13.10	12.31/18.55/15.16	12.38/24.25/18.96	85.96/185.07/180.88
1e-07	35.94/43.35/30.00	24.92/30.18/23.45	17.98/20.89/17.45	14.79/18.90/15.52	12.10/15.47/13.35	15.48/22.00/17.84	14.86/31.16/26.21	411.89/620.52/652.55

Table E.11: OPT ppl on wikitext/opt/c4 with W4<sup>asym</sup>-A8<sup>sym</sup>/A8<sup>asym</sup> and ZQ-Global.

Precision	125m	350m	1.3b	2.7b	6.7b	13b	30b	66b
W4 <sup>asym</sup> -A8 <sup>sym</sup> Block								
0.001	34.90/44.82/28.27	28.08/56.82/33.84/69	nan/nan/nan	18290.61/9784.37/12099.01	16014.50/8655.69/12304.55	248961.98/4882.70/104880.55	5667.05/23709.03/33007.17	29782.43/20410.10/23559.66
0.0005	31.78/38.56/27.20	29.24/54.15/29.76	10610.96/9438.99/6752.84	12499.29/8111.26/10677.01	1235.59/11476.10/10223.89	74731.13/4494.68/29286.49	5187.73/28548.95/23056.78	18717.63/11744.97/12903.33
0.0001	32.92/38.86/27.23	24.11/29.20/22.47	16.02/19.20/16.92	17.65/16.55/17.13	12305.59/11476.10/10223.89	1603.04/1041.12/1219.52	2235.59/2182.45/2187.15	7926.43/1040.45/4805.98
5e-05	32.16/37.54/27.23	24.15/29.87/24.46	16.02/19.20/16.92	17.65/16.55/17.13	1241.10/11476.10/10223.89	1329.55/1781.73/1767.30	9512.41/4543.45/5373.00	NAN
1e-05	32.57/38.43/27.53	24.01/28.81/22.57	16.12/19.15/16.23	13.98/17.70/14.87	99.27/118.19/88.74	529.82/361.44/256.46	1936.12/1388.68/947.45	10077.70/9296.31/11462.28
5e-06	32.83/38.37/27.71	24.13/29.30/22.68	16.45/19.64/16.57	14.42/18.01/15.27	70.26/62.28/54.47	373.82/694.33/170.40	820.90/847.19/543.59	1867.57/1878.76/4117.49
1e-06	34.79/41.79/29.30	24.68/30.01/23.23	17.90/21.94/18.01	14.83/18.63/15.70	38.27/39.77/52.26	117.83/141.63/96.83	261.19/84.44/272.0	1500.51/1275.54/169.50
5e-07	Nan	Nan	Nan	Nan	Nan	Nan	233.71/700.40/337.15	1715.98/1546.50/1799.35
1e-07	Nan	Nan	Nan	Nan	Nan	Nan	913.95/1117.58/1065.87	2012.91/1917.48/1817.92
W4 <sup>asym</sup> -A8 <sup>asym</sup> Block								
0.001	37.80/47.88/34.43	9004.01/5636.90/1386.96	12638.86/1016.64	11288.86/6474.19/1000.00	12324.01/953.43/4889.36	13296.67/4765.36/5026.76	32393.54/1374.97/1041.59	4828.52/3048.89/42797.96
0.0005	32.65/38.86/27.23	28.40/36.94/24.68	16.31/19.71/16.44	43.76/29.11/33.35	22824.00/1530.32/1623.28	22024.01/13926.04/11130.94	10171.49/2309.78/7954.12	NAN
0.0001	31.61/37.00/27.10	24.64/29.13/22.28	16.31/19.71/16.44	43.76/29.11/33.35	22024.01/13926.04/11130.94	10171.49/2309.78/7954.12	18693.08/11639.42/10798.26	nan/nan/nan
5e-05	32.21/37.46/27.18	23.66/28.56/22.21	16.02/19.02/15.92	13.48/17.57/14.24	839.48/213.76/286.05	1035.13/14.01/1472.08	8085.92/3545.21/4933.07	nan/nan/nan
1e-05	32.35/38.21/27.38	23.59/28.66/22.24	15.77/18.61/15.83	13.09/16.56/14.00	12.09/14.69/12.90	11.50/15.01/12.41	13.76/22.87/15.72	9745.58/1557.05/1039.65
5e-06	32.59/38.49/27.68	23.62/28.63/22.33	15.78/18.80/15.95	13.23/16.65/14.12	12.03/14.60/12.86	12.72/16.31/13.20	12.94/17.61/13.41	83.35/137.83/128.11
1e-06	34.68/41.56/29.26	24.08/29.21/22.68	16.66/20.03/16.69	13.30/16.74/14.33	12.43/15.52/13.36	12.28/16.13/13.19	16.00/19.60/14.88	31.51/58.00/23.95
5e-07	Nan	Nan	Nan	Nan	Nan	Nan	Nan	31.09/73.23/24.44
1e-07	Nan	Nan	Nan	Nan	Nan	Nan	Nan	241.81/541.81/565.58

Table E.12: BLOOM ppl on wikitext/opt/c4 with W4<sup>asym</sup>-A8<sup>sym</sup>/A8<sup>asym</sup>. See Table E.13 for all learning rate results of ZQ-Local and Table E.14 of ZQ-Global.

Precision	560m	1.1b	1.7b	3b	7.1b	176b
W4 <sup>asym</sup> -A8 <sup>sym</sup> Block						
RTN	25.56/47.53/27.31	24.80/70.99/26.71	17.36/31.95/19.89	14.82/25.63/17.47	12.33/21.62/15.13	9.12/15.58/14.04
GPTQ	24.13/44.79/25.86	25.69/68.65/27.08	16.63/30.54/19.12	14.18/24.42/16.82	12.04/21.07/14.75	8.92/15.16/13.56
ZQ-Local*	24.45/45.73/26.22	19.50/52.67/21.73	16.71/30.23/19.09	14.37/24.72/16.99	12.00/20.79/14.78	8.52/14.29/11.41
ZQ-Global*	23.93/44.31/25.68	19.71/51.98/21.85	16.34/29.36/18.82	14.13/24.34/16.76	11.84/20.58/14.59	8.76/14.60/11.68
W4 <sup>asym</sup> -A8 <sup>asym</sup> Block						
RTN	25.37/46.99/27.16	24.08/68.95/26.17	17.12/31.46/19.67	14.74/25.38/17.37	12.22/21.36/15.00	8.73/15.10/12.83
GPTQ	24.09/44.29/25.66	24.50/67.37/26.62	16.39/29.83/18.91	14.13/24.47/16.73	11.91/20.72/14.62	8.55/14.74/12.31
ZQ-Local*	24.29/45.19/26.10	19.13/52.89/21.63	16.54/30.11/18.92	14.32/24.73/16.94	11.94/20.63/14.68	8.33/14.01/11.22
ZQ-Global*	23.86/44.16/25.62	19.54/51.72/21.79	16.23/29.40/18.68	14.15/24.29/16.72</td		

Table E.13: BLOOM ppl on wikitext/opt/c4 with W4<sup>asym</sup>-A8<sup>sym</sup>/A8<sup>asym</sup> and ZQ-Local.

Precision	560m	1.1b	1.7b	3b	7.1b	176b
W4 <sup>asym</sup> -A8 <sup>sym</sup> Block						
0.001	25.51/47.89/27.15	19.73/54.63/22.18	16.96/31.47/19.44	14.59/25.69/17.32	12.51/21.85/15.34	8.62/14.42/11.50
0.0005	25.18/47.35/26.95	19.62/53.64/22.03	16.98/31.75/19.47	14.52/25.22/17.18	12.03/21.01/14.82	8.59/14.38/11.45
0.0001	24.79/46.37/26.44	19.50/52.67/21.73	16.68/30.51/19.18	14.44/25.12/17.05	12.00/20.79/14.78	8.52/14.29/11.41
5e-05	24.56/46.29/26.34	23.93/69.17/26.19	16.71/30.23/19.09	14.37/24.72/16.99	12.05/20.92/14.82	8.55/14.34/11.44
1e-05	24.45/45.73/26.22	23.65/66.73/25.80	16.66/30.69/19.16	14.40/24.94/17.02	12.12/21.14/14.86	8.65/14.97/12.01
5e-06	24.48/45.66/26.33	23.87/67.26/25.84	16.78/30.72/19.23	14.44/24.91/17.07	12.15/21.23/14.88	8.70/15.04/12.37
1e-06	24.91/46.35/26.72	24.09/68.13/26.05	17.03/31.28/19.52	14.60/25.18/17.24	12.22/21.31/14.99	8.91/15.25/13.35
W4 <sup>asym</sup> -A8 <sup>asym</sup> Block						
0.001	25.26/46.43/26.98	19.69/54.26/22.14	16.88/32.16/19.40	15.15/26.58/17.76	12.40/22.29/15.28	8.40/14.06/11.26
0.0005	24.89/47.99/26.82	19.54/53.57/21.98	16.73/31.02/19.29	14.50/25.52/17.11	11.94/20.70/14.76	8.33/14.01/11.22
0.0001	24.60/45.75/26.44	19.13/52.89/21.63	16.54/30.36/19.10	14.37/24.91/16.93	11.94/20.63/14.68	8.35/14.04/11.24
5e-05	24.41/45.08/26.23	23.59/67.14/25.79	16.54/30.11/18.92	14.29/24.83/16.92	11.95/20.71/14.71	8.36/14.10/11.25
1e-05	24.29/45.19/26.10	23.35/65.26/25.38	16.51/30.20/19.00	14.32/24.73/16.94	11.97/20.93/14.74	8.44/14.30/11.45
5e-06	24.31/45.25/26.15	23.41/66.18/25.48	16.63/30.37/19.09	14.33/24.74/16.96	12.03/20.95/14.78	8.52/14.66/11.86
1e-06	24.76/45.92/26.62	23.52/66.38/25.66	16.81/30.71/19.30	14.53/24.92/17.14	12.10/21.07/14.87	8.62/14.92/12.41

 Table E.14: BLOOM ppl on wikitext/opt/c4 with W4<sup>asym</sup>-A8<sup>sym</sup>/A8<sup>asym</sup> and ZQ-Global.

Precision	560m	1.1b	1.7b	3b	7.1b	176b
W4 <sup>asym</sup> -A8 <sup>sym</sup> Block						
0.001	174250016.00/201477664.00/1348168.88	423532.56/906908.06/322995.69	573201.81/1089364.38/498071.91	544376.56/696942.56/540949.06	nan/nan/nan	NaN
0.0005	70978.52/29214230.00/1151.72	2880.81/15732.60/309.13	505479.44/629035.56/29283.36	140595.53/181082.25/33785.79	378033.53/789890.00/191543.91	NaN
0.0001	24.04/45.38/25.83	19.44/53.38/21.77	16.34/29.36/18.82	14.32/24.74/16.88	12.12/22.00/14.80	249.47/26690.76/26.96
5e-05	23.93/44.31/25.68	19.71/51.98/21.85	16.18/29.71/18.71	14.13/24.34/16.76	11.84/20.58/14.59	9.00/15.57/11.61
1e-05	23.99/44.44/25.77	22.75/58.31/23.63	16.28/29.96/18.81	14.29/24.53/16.87	11.87/20.57/14.64	8.76/14.60/11.68
5e-06	24.14/44.77/25.90	23.90/64.81/25.29	16.36/30.33/18.91	14.32/24.68/16.95	11.91/20.60/14.71	9.07/15.12/11.98
1e-06	24.62/45.70/26.33	25.55/71.49/27.44	16.61/30.47/19.17	14.51/24.91/17.11	12.06/20.93/14.86	11.25/19.93/15.76
W4 <sup>asym</sup> -A8 <sup>asym</sup> Block						
0.001	9050902.00/2932002.50/131873960.00	499829.19/393190.53/346682.47	1260531.12/2019747.88/460627.16	1022130.19/872164.88/679662.62	nan/nan/nan	NaN
0.0005	7633.14/378055.53/1032.16	4271.83/58547.50/1555.66	87087.04/217513.30/37000.13	575008.56/814032.50/230285.80	1212241.00/238940.25/150466.50	NaN
0.0001	23.96/45.36/25.80	19.37/52.25/21.88	16.29/29.36/18.81	14.32/24.66/16.86	12.05/22.30/14.77	1400.84/11880.12/392.79
5e-05	23.86/44.16/25.62	19.54/51.72/21.79	16.23/29.40/18.68	14.15/24.29/16.72	11.82/20.44/14.54	8.73/20.30/11.41
1e-05	23.96/44.24/25.72	22.55/58.10/23.49	16.27/29.82/18.78	14.16/24.35/16.80	11.80/20.37/14.56	8.62/14.40/11.49
5e-06	24.01/44.68/25.83	23.67/64.20/25.08	16.30/29.96/18.85	14.24/24.49/16.86	11.81/20.50/14.60	8.69/14.56/11.58
1e-06	24.53/45.60/26.26	24.82/71.17/26.84	16.55/30.35/19.10	14.40/24.76/17.01	11.97/20.83/14.77	9.14/16.63/17.69

Table E.15: OPT full results of Table 4.

Method	125m	350m	1.3b	2.7b	6.7b	13b	30b	66b
BS=1024								
RTN	N/A	25.42/30.62/23.61	16.90/19.78/16.59	N/A	11.63/14.41/12.65	10.47/13.09/11.75	9.97/12.40/11.09	9.83/12.31/10.77
	N/A	26.55	17.76	N/A	12.90	11.77	11.15	10.97
GPTQ	N/A	23.65/29.09/22.43	15.16/18.00/15.34	N/A	11.10/13.40/11.99	10.28/12.49/11.29	9.58/11.91/10.75	9.56/11.61/10.44
	N/A	25.05	16.17	N/A	12.16	11.36	10.75	10.54
ZQ-Global*	N/A	23.27/27.97/21.93	12.93/15.90/13.64	N/A	10.98/13.60/12.04	10.33/12.69/11.50	9.78/12.16/10.90	9.52/11.58/10.46
	N/A	24.39	16.18	N/A	12.21	11.50	10.95	10.52
BS=512								
RTN	N/A	25.05/29.74/23.21	15.71/19.05/16.09	13.67/16.93/14.23	11.32/14.22/12.50	10.45/12.99/11.68	10.03/12.27/11.03	9.83/12.15/10.67
	N/A	26.00	16.95	14.94	12.68	11.71	11.11	10.89
GPTQ	N/A	23.33/28.48/22.13	15.15/17.95/15.26	12.65/15.61/13.53	10.94/13.37/11.94	10.18/12.49/11.29	9.58/11.87/10.75	9.53/11.59/10.43
	N/A	24.65	16.12	13.93	12.08	11.32	10.73	10.52
ZQ-Global*	N/A	23.41/27.67/21.92	14.91/17.97/15.25	12.92/15.59/13.55	11.08/13.51/11.99	10.29/12.68/11.46	9.79/12.16/10.87	9.51/11.65/10.44
	N/A	24.34	15.97	14.02	12.19	11.48	10.94	10.53
BS=256								
RTN	31.62/38.19/27.62	24.76/29.44/22.96	15.54/18.96/15.90	13.56/16.62/14.02	11.19/14.12/12.40	10.39/12.93/11.61	9.95/12.24/10.98	9.70/12.09/10.62
	32.48	25.72	16.80	14.73	12.57	11.64	11.06	10.80
GPTQ	30.56/37.20/26.68	23.37/28.33/21.97	14.95/17.63/15.16	12.59/15.60/13.49	10.93/13.29/11.92	10.15/12.43/11.27	9.58/11.91/10.74	9.49/11.60/10.40
	31.48	24.56	15.91	13.89	12.05	11.28	10.74	10.50
ZQ-Global*	30.45/35.35/26.24	23.06/27.72/21.74	14.93/17.45/15.15	12.99/15.47/13.50	10.96/13.45/12.00	10.25/12.61/11.43	9.73/12.14/10.89	9.49/11.58/10.42
	30.68	24.17	15.84	13.99	12.14	11.43	10.92	10.50
BS=128								
RTN	30.62/36.67/27.10	24.12/29.34/22.70	15.35/18.52/15.66	13.19/16.24/13.88	11.11/13.94/12.28	10.31/12.82/11.54	9.93/12.12/10.93	9.56/11.85/10.56
	31.47	25.39	16.51	14.43	12.44	11.56	11.00	10.65
GPTQ	30.76/36.13/26.52	23.29/27.94/21.98	14.93/17.95/15.10	12.49/15.59/13.46	10.87/13.34/11.90	10.11/12.47/11.27	9.60/11.88/10.73	9.44/11.53/10.40
	31.14	24.40	15.85	13.85	12.03	11.28	10.74	10.45
ZQ-Global*	29.52/34.63/25.98	22.78/27.56/21.65	15.02/17.00/15.07	12.67/15.37/13.45	10.92/13.42/11.96	10.16/12.61/11.41	9.74/12.01/10.82	9.43/11.49/10.40
	30.04	23.99	15.86	13.83	12.10	11.39	10.86	10.44
BS=64								
RTN	30.74/36.68/26.87	24.28/28.95/22.59	15.21/18.15/15.47	13.20/16.13/13.75	11.01/13.71/12.17	10.27/12.79/11.49	9.82/12.05/10.89	9.46/11.70/10.49
	31.43	25.27	16.28	14.36	12.30	11.52	10.92	10.55
GPTQ	30.25/35.72/26.43	23.39/27.55/21.75	14.81/17.40/15.06	12.54/15.54/13.44	10.87/13.29/11.89	10.09/12.44/11.27	9.55/11.89/10.72	9.33/11.49/10.38
	30.80	24.23	15.76	13.84	12.02	11.27	10.72	10.40
ZQ-Global*	29.69/34.24/25.72	22.94/27.49/21.54	14.90/17.43/15.01	12.80/15.47/13.44	10.92/13.33/11.93	10.21/12.58/11.38	9.69/12.01/10.81	9.41/11.49/10.39
	29.88	23.99	15.78	13.90	12.06	11.39	10.84	10.43
BS=32								
RTN	30.48/36.32/26.64	23.88/28.66/22.36	14.99/17.87/15.32	12.89/16.00/13.67	10.89/13.70/12.13	10.32/12.73/11.45	9.76/12.00/10.85	9.56/11.55/10.44
	31.14	24.97	16.06	14.18	12.24	11.50	10.87	10.52
GPTQ	29.13/34.89/25.90	23.09/27.59/21.65	14.80/17.41/15.04	12.45/15.55/13.42	10.89/13.32/11.89	10.08/12.48/11.27	9.51/11.92/10.73	Diverge
	29.97	24.11	15.75	13.81	12.03	11.28	10.72	Diverge
ZQ-Global*	28.93/34.29/25.63	22.85/27.23/21.50	14.80/17.34/14.99	12.74/15.32/13.40	10.82/13.36/11.91	10.23/12.61/11.37	9.68/11.95/10.80	9.37/11.47/10.38
	29.62	23.86	15.71	13.82	12.03	11.41	10.81	10.41

Table E.16: BLOOM W4<sup>asym</sup>-A16 with various block-size out of the best result from GPTQ and ZQ-Global.

Method	560m	1.1b	1.7b	3b	7.1b	176b
<b>BS=1024</b>						
RTN	24.90/46.37/26.68 32.65	N/A N/A	16.57/30.14/19.00 21.90	N/A N/A	1019.51/1351.45/601.35 990.77	53.41/160.05/43.64 85.70
GPTQ	23.90/43.99/25.47 31.12	N/A N/A	16.12/29.13/18.61 21.29	N/A N/A	11.57/19.82/14.33 15.24	8.16/13.70/11.02 10.96
ZQ-Global	23.62/43.90/25.41 30.98	N/A N/A	15.98/28.67/18.44 21.03	N/A N/A	11.91/20.84/14.58 15.78	8.23/13.94/11.09 11.09
<b>BS=512</b>						
RTN	24.78/46.07/26.45 32.44	19.41/53.64/21.85 31.63	16.47/29.84/18.88 21.73	14.29/24.84/17.05 18.73	142.38/314.10/100.09 185.52	33.88/103.57/31.02 56.16
GPTQ	23.63/43.96/25.36 30.98	18.52/49.73/20.91 29.72	16.07/29.87/18.50 21.48	13.79/23.77/16.41 17.99	11.54/19.75/14.30 15.20	8.14/13.70/11.02 10.95
ZQ-Global	23.50/43.53/25.23 30.75	18.31/49.06/20.82 29.40	15.93/28.47/18.38 20.93	13.82/23.92/16.47 18.07	11.85/20.17/14.42 15.48	8.20/13.86/11.07 11.04
<b>BS=256</b>						
RTN	24.09/45.13/26.02 31.75	18.87/52.29/21.44 30.87	16.27/29.72/18.76 21.58	14.16/24.42/16.90 18.49	121.09/281.67/88.59 163.78	12.55/27.29/15.60 18.48
GPTQ	23.31/43.43/25.12 30.62	18.36/49.13/20.79 29.42	16.07/29.10/18.46 21.21	13.76/23.61/16.38 17.92	11.55/19.72/14.29 15.18	8.14/13.70/11.01 10.95
ZQ-Global	23.17/43.16/25.13 30.49	18.24/48.78/20.75 29.26	15.81/28.71/18.32 20.95	13.79/23.69/16.42 17.97	11.59/19.92/14.36 15.29	8.17/13.80/11.06 11.01
<b>BS=128</b>						
RTN	23.82/44.78/25.75 31.45	18.62/51.31/21.17 30.37	16.13/29.89/18.66 21.56	14.00/24.19/16.71 18.30	23.90/49.80/24.15 32.62	8.84/15.62/11.70 12.06
GPTQ	23.27/43.10/24.99 30.45	18.14/48.72/20.73 29.20	16.03/28.96/18.41 21.13	13.72/23.65/16.34 17.90	11.52/19.73/14.26 15.17	8.14/13.67/11.01 10.94
ZQ-Global	23.14/42.95/24.97 30.35	18.17/48.53/20.70 29.13	15.75/28.71/18.29 20.92	13.73/23.65/16.37 17.92	11.56/19.77/14.32 15.22	8.17/13.78/11.03 10.99
<b>BS=64</b>						
RTN	23.65/44.04/25.51 31.07	18.53/50.02/21.03 29.86	16.06/29.57/18.60 21.41	13.93/23.95/16.60 18.16	11.85/20.51/14.65 15.67	8.31/14.14/11.18 11.21
GPTQ	23.11/42.95/24.94 30.33	18.14/48.87/20.65 29.22	16.00/28.91/18.38 21.10	13.72/23.68/16.33 17.91	11.51/19.70/14.27 15.16	8.14/13.69/11.00 10.94
ZQ-Global	23.00/42.80/24.91 30.24	18.10/48.30/20.64 29.01	15.68/28.55/18.25 20.82	13.70/23.63/16.36 17.90	11.53/19.67/14.27 15.16	8.17/13.72/11.02 10.97
<b>BS=32</b>						
RTN	23.60/43.91/25.50 31.00	18.63/50.13/21.04 29.93	15.98/29.56/18.56 21.37	13.92/23.90/16.53 18.12	11.65/20.01/14.43 15.36	8.20/13.86/11.07 11.04
GPTQ	23.10/43.19/24.91 30.40	18.17/48.35/20.66 29.06	15.95/28.95/18.36 21.08	13.76/23.60/16.33 17.89	11.53/19.71/14.27 15.17	8.14/13.70/11.00 10.95
ZQ-Global	23.07/42.63/24.82 30.18	18.07/48.07/20.59 28.91	15.66/28.58/18.21 20.82	13.72/23.59/16.33 17.88	11.52/19.71/14.26 15.16	8.16/13.69/11.01 10.95

Table E.17: OPT full results of three-bit weight with various block-size.

Method	125m	350m	1.3b	2.7b	6.7b	13b	30b	66b
Full Row								
RTN	2005.20/1848.83/1222.00	47.43/53.38/36.93	4399.18/4400.98/3551.88	8326.78/4208.57/4895.83	878.00/735.86/910.10	1953.43/1953.60/1669.76	439.39/691.94/437.96	1465.06/1564.59/1282.58
	1722.01	45.91	4117.35	5810.40	841.32	1858.93	523.09	1437.41
GPTQ	845.81/599.71/496.14	30.65/34.09/26.15	20.23/27.39/19.45	15.91/19.26/16.01	12.69/15.90/13.96	11.36/13.71/12.21	10.10/12.54/11.20	16.77/21.16/15.39
ZQ-Global*	46.47/58.55/35.45	29.64/36.51/25.55	32.48/94.57/28.97	60.91/116.22/36.45	23.87/29.75/23.88	44.70/60.78/46.18	13.16/20.49/13.48	28.93/75.91/27.28
	46.82	30.57	52.01	71.19	25.83	50.55	15.71	44.04
BS=1024								
RTN	N/A	44.57/49.58/35.09	1950.00/2317.55/1913.55	3810.79/2563.06/3054.91	50.01/70.17/99.21	265.62/417.03/261.93	362.47/252.33/364.45	523.81/846.60/1021.17
	N/A	43.08	2060.37	3142.92	73.13	314.86	326.42	797.20
GPTQ	N/A	29.78/33.34/25.66	19.03/23.32/18.14	N/A	11.69/14.31/12.70	10.56/12.96/11.70	9.89/12.19/11.02	12.84/16.17/13.02
ZQ-Global*	N/A	29.19/34.57/25.11	19.83/29.77/19.79	N/A	12.90	11.74	11.43	14.01
	N/A	29.62	23.13	N/A	13.99/18.82/14.76	13.43/19.28/13.76	11.10/14.46/11.94	11.87/14.86/12.13
BS=512								
RTN	N/A	37.74/45.10/31.85	1777.53/1304.55/852.03	1604.07/1407.49/1487.78	25.13/40.56/40.08	130.75/175.33/135.67	620.53/340.68/416.28	198.01/457.78/426.15
	N/A	38.23	1311.37	1499.78	35.26	147.25	459.16	360.65
GPTQ	N/A	28.46/32.54/25.14	18.02/21.35/17.46	14.38/17.24/14.79	11.57/14.33/12.57	10.41/12.97/11.64	9.77/12.18/10.97	11.89/14.48/12.40
ZQ-Global*	N/A	28.71	18.94	15.47	12.82	11.67	10.97	12.92
	N/A	27.81/33.57/24.55	18.31/23.54/17.99	18.10/29.47/17.15	12.54/16.60/13.62	11.82/15.98/12.81	10.48/13.36/11.66	11.26/13.95/11.79
	N/A	28.65	19.95	21.57	14.25	13.54	11.83	12.33
BS=256								
RTN	4349.14/2907.61/2510.75	35.36/42.07/30.81	127.17/358.19/142.49	670.51/550.66/531.80	19.10/32.39/27.26	42.52/56.35/43.32	32.84/60.38/33.48	210.01/478.13/413.00
	3255.84	36.08	209.28	584.32	26.25	47.40	42.23	367.05
GPTQ	41.81/49.95/32.48	27.60/33.73/24.88	16.97/20.19/16.70	13.69/17.06/14.54	11.65/14.24/12.48	10.35/12.93/11.61	9.66/12.10/10.93	11.60/13.98/11.92
ZQ-Global*	41.41	28.74	17.95	15.10	12.70	11.63	10.90	12.50
	38.60/46.57/31.36	26.88/32.79/24.08	16.82/21.21/17.05	14.86/19.63/15.37	11.86/15.87/13.10	11.33/14.95/12.48	10.41/12.95/11.41	10.26/12.66/11.08
	38.85	27.92	18.36	16.62	13.61	12.92	11.59	11.34
BS=128								
RTN	3446.89/2156.26/1484.15	33.13/41.23/29.51	49.40/88.45/45.07	153.68/155.21/113.98	16.34/26.86/21.98	17.80/25.95/18.28	45.83/43.91/57.50	106.84/241.02/212.94
	2362.43	34.62	60.97	140.96	21.72	20.67	49.08	186.93
GPTQ	40.00/45.73/31.15	27.68/34.04/25.18	16.47/19.90/16.47	13.81/16.96/14.37	11.57/14.10/12.41	10.35/12.84/11.58	9.73/12.08/10.91	10.96/13.27/11.45
ZQ-Global*	38.96	28.97	17.61	15.05	12.69	11.59	10.91	11.90
	36.57/43.88/29.94	25.75/31.59/23.57	16.28/20.20/16.67	14.27/18.41/14.90	11.70/15.05/12.68	11.13/15.07/12.17	10.31/12.99/11.32	10.12/12.66/11.01
	36.80	26.97	17.72	15.86	13.14	12.79	11.54	11.27
BS=64								
RTN	708.02/477.13/287.03	32.61/42.14/29.09	25.43/38.84/24.63	72.84/69.27/48.07	14.11/21.71/16.56	14.13/20.08/15.25	20.55/32.74/24.49	30.66/70.73/65.57
	490.73	34.61	29.63	63.39	17.46	16.48	25.93	55.65
GPTQ	37.15/42.59/30.07	27.68/33.55/25.12	16.25/19.80/16.32	13.66/16.69/14.37	11.42/13.98/12.37	10.37/12.90/11.58	9.68/12.17/10.92	10.39/12.65/11.15
ZQ-Global*	36.60	28.78	17.46	14.91	12.59	11.62	10.92	11.40
	35.82/40.98/29.65	25.31/31.60/23.38	16.05/19.77/16.39	13.33/16.92/14.31	11.56/14.70/12.59	10.88/13.64/12.04	10.04/12.70/11.27	10.04/12.06/10.81
	35.48	26.76	17.40	14.85	12.95	12.19	11.34	10.97
BS=32								
RTN	72.83/88.62/54.25	32.36/40.76/29.06	20.22/27.31/19.81	31.12/42.01/26.83	13.38/18.56/15.44	13.06/18.35/14.38	11.12/15.05/12.35	19.29/43.61/34.10
	71.90	34.06	22.44	33.32	15.79	15.26	12.84	32.33
GPTQ	38.26/45.01/30.92	27.16/33.65/24.97	16.13/19.83/16.45	13.66/17.06/14.50	11.43/14.08/12.42	10.48/12.96/11.65	9.78/12.24/10.96	Diverge
ZQ-Global*	33.44/39.48/28.33	25.19/30.73/23.22	15.62/19.52/16.20	13.35/16.64/14.18	11.56/14.38/12.61	10.86/13.64/12.03	10.25/12.86/11.28	9.99/12.05/10.81
	33.75	26.98	17.11	14.73	12.85	12.17	11.46	10.95

Table E.18: BLOOM W3<sup>asym</sup>-A16 with various block-size out of the best result from GPTQ and ZQ-Global.

Method	560m	1.1b	1.7b	3b	7.1b	176b
Full row						
RTN	68.45/132.83/59.22 86.83	118.61/317.41/99.65 178.56	31.15/67.23/34.02 44.14	31.07/59.03/32.17 40.76	66140.72/78568.16/44504.19 63071.02	100371.84/166012.19/137892.34 134758.79
GPTQ	46.92/84.69/39.50 57.04	49.78/142.95/43.84 78.85	19.70/41.35/21.74 27.59	22.84/46.49/22.90 30.74	52966.59/52979.88/37115.48 47687.32	Diverge Diverge
ZQ-Global	33.20/64.61/32.30 43.37	34.16/100.05/29.22 54.48	19.22/36.30/21.25 25.59	18.41/33.10/20.79 24.10	273.55/439.59/100.79 271.31	27.19/75.74/45.45 49.46
BS=1024						
RTN	47.00/86.57/43.37 58.98	70.81/230.74/70.78 124.11	35.41/65.75/33.54 44.90	22.12/40.65/24.55 29.11	25654.77/25531.66/15868.46 22351.63	141324.41/183583.73/200436.33 175114.82
GPTQ	31.25/58.80/30.94 40.33	N/A N/A	19.11/37.07/20.90 25.69	N/A N/A	12.59/21.95/15.21 16.58	8.31/13.96/11.17 11.15
ZQ-Global	28.91/55.81/29.59 38.10	N/A N/A	18.20/34.13/20.40 24.24	N/A N/A	30.94/119.98/21.39 57.44	15.98/32.85/19.85 22.89
BS=512						
RTN	41.58/79.83/39.41 53.61	33.83/116.88/37.34 62.68	25.95/49.65/26.77 34.12	19.94/38.58/22.58 27.03	9777.49/8000.29/5407.46 7728.41	202051.34/273707.81/279776.97 251845.38
GPTQ	28.08/53.15/29.05 36.76	21.20/61.42/23.33 35.32	18.41/34.47/20.43 24.44	15.08/26.14/17.53 19.58	12.32/21.29/15.01 16.21	8.30/13.98/11.16 11.15
ZQ-Global	26.80/50.49/28.31 35.20	20.77/57.57/22.89 33.75	17.64/33.19/19.91 23.58	15.16/26.51/17.57 19.75	16.35/28.75/15.76 20.29	11.38/20.36/14.66 15.47
BS=256						
RTN	36.13/70.37/36.29 47.60	28.65/95.72/31.80 52.06	21.67/42.59/23.80 29.35	17.64/32.82/20.69 23.72	1322.61/1864.55/946.92 1378.02	166006.80/187829.98/198052.83 183963.20
GPTQ	27.10/51.11/28.24 35.48	20.60/56.57/22.77 33.31	17.97/33.28/20.04 23.76	14.82/25.79/17.31 19.31	12.27/21.24/14.93 16.15	8.27/13.99/11.14 11.13
ZQ-Global	25.96/49.75/27.59 34.43	20.21/54.83/22.33 32.46	17.43/32.14/19.67 23.08	14.85/25.79/17.33 19.32	12.85/22.00/15.04 16.63	9.07/15.88/11.88 12.28
BS=128						
RTN	34.71/66.56/35.27 45.51	24.43/73.77/26.90 41.70	19.59/37.22/21.98 26.26	16.11/28.81/18.89 21.27	108.32/252.15/74.42 144.96	111057.84/101926.99/105339.26 106108.03
GPTQ	26.29/49.86/27.54 34.56	20.26/55.76/22.42 32.81	17.77/32.65/19.92 23.45	14.58/25.25/17.11 18.98	12.18/21.06/14.86 16.03	8.26/13.92/11.12 11.10
ZQ-Global	25.28/48.24/26.96 33.49	19.79/54.04/22.03 31.95	17.12/31.42/19.31 22.62	14.62/25.73/17.17 19.17	12.04/21.02/14.82 15.96	8.43/14.44/11.29 11.39
BS=64						
RTN	30.88/59.01/32.08 40.66	23.04/67.93/25.49 38.82	19.35/37.67/21.80 26.27	15.64/27.56/18.39 20.53	37.15/65.22/33.22 45.20	198.66/488.11/128.62 271.80
GPTQ	26.31/49.91/27.17 34.46	20.11/55.06/22.23 32.47	17.94/32.42/19.76 23.37	14.62/25.39/17.07 19.02	12.13/21.07/14.83 16.01	8.26/13.93/11.11 11.10
ZQ-Global	25.17/48.01/26.59 33.26	19.51/53.27/21.75 31.51	16.88/31.14/19.22 22.41	14.51/25.18/17.05 18.91	12.00/20.85/14.74 15.86	8.35/14.06/11.20 11.21
BS=32						
RTN	30.15/57.55/31.51 39.74	23.49/70.15/25.56 39.73	18.96/36.54/21.42 25.64	15.56/27.48/18.32 20.46	13.06/23.77/16.05 17.62	10.28/18.90/13.27 14.15
GPTQ	25.96/49.99/27.06 34.33	19.97/54.79/22.16 32.31	17.60/32.24/19.76 23.20	14.55/25.76/17.06 19.12	12.20/21.01/14.85 16.02	8.28/13.95/11.13 11.12
ZQ-Global	25.09/47.36/26.34 32.93	19.43/52.95/21.64 31.34	16.86/30.49/19.11 22.15	14.50/25.36/16.99 18.95	12.00/20.84/14.72 15.85	8.35/14.04/11.20 11.20

Table E.19: Full results of BLOOM-176B with different quantization bits

Bits	3	4	5	6	7	8
Per-row	27.19/75.74/45.45	8.16/13.70/11.02	8.13/13.67/10.99	8.11/13.63/10.98	8.11/13.62/10.97	8.10/13.62/10.98
1024	8.31/13.96/11.17	8.14/13.70/11.02	8.11/13.62/10.97	8.11/13.62/10.97	8.11/13.63/10.97	N/A
64	8.26/13.93/11.11	8.14/13.69/11.00	8.11/13.62/10.96	N/A	N/A	N/A

Table E.20: OPT full results of Table 5.

Method	125m	350m	1.3b	2.7b	6.7b	13b	30b	66b
W4 <sup>asym</sup> full row and A8 <sup>sym</sup> 128								
RTN	36.64/44.84/30.90 37.46	25.58/31.06/23.99 26.88	19.96/22.31/18.20 20.16	18.42/23.01/18.56 20.00	12.04/15.92/13.20 13.72	10.79/13.65/12.11 12.18	10.10/13.17/11.37 11.54	20.50/45.58/25.37 30.48
GPTQ	31.82/38.82/27.54 32.73	23.78/28.96/22.61 25.12	15.56/18.27/15.62 16.48	13.02/15.88/13.76 14.22	11.22/13.59/12.11 12.31	10.25/12.65/11.37 11.42	9.56/11.94/10.79 10.76	9.62/11.72/10.54 10.63
ZQ-Local								9.79/11.94/10.65 10.79
ZQ-Global	31.69/36.66/27.19 31.85	23.47/28.18/22.03 24.56	15.53/18.35/15.73 16.54	13.02/16.11/13.82 14.32	11.29/13.70/12.19 12.39	10.43/12.91/11.64 11.66	9.86/12.28/11.00 11.05	9.62/11.84/10.63 10.70
W4 <sup>asym</sup> 128 and A8 <sup>sym</sup> 128								
RTN	30.61/36.57/27.08 31.42	24.14/29.47/22.80 25.47	15.46/18.68/15.77 16.64	13.24/16.36/13.95 14.52	11.16/14.08/12.35 12.53	10.35/12.89/11.57 11.60	9.95/12.15/10.95 11.02	9.58/11.90/10.58 10.69
GPTQ	30.47/36.45/26.45 31.12	23.43/28.12/22.06 24.54	14.90/17.62/15.17 15.90	12.51/15.63/13.48 13.87	10.88/13.35/11.93 12.05	10.17/12.48/11.28 11.31	9.58/11.86/10.74 10.73	9.35/11.54/10.40 10.43
ZQ-Local								9.40/11.63/10.51 10.51
ZQ-Global	29.59/34.68/25.91 30.06	22.59/27.93/21.68 24.07	14.87/17.55/15.11 15.84	12.65/15.45/13.48 13.86	10.88/13.40/11.94 12.08	10.20/12.67/11.43 11.43	9.74/12.03/10.83 10.87	9.40/11.51/10.42 10.44
W4 <sup>asym</sup> full row and A8 <sup>asym</sup> 128								
RTN	36.61/44.71/30.85 37.39	25.50/30.93/23.88 26.77	19.58/22.08/18.01 19.89	19.53/24.38/19.68 21.20	11.91/15.35/13.01 13.42	10.68/13.50/12.02 12.07	10.13/13.21/11.37 11.57	17.90/32.15/20.02 23.36
GPTQ	32.15/39.58/27.65 33.13	23.48/28.92/22.46 24.95	15.43/18.24/15.55 16.40	12.92/15.94/13.74 14.20	11.17/13.59/12.09 12.29	10.35/12.63/11.36 11.45	9.65/11.95/10.79 10.80	9.58/11.71/10.55 10.61
ZQ-Local								10.05/11.91/10.61 10.86
ZQ-Global	31.55/37.49/27.25 32.10	23.34/28.33/22.08 24.58	15.52/18.55/15.61 16.56	13.07/16.09/13.82 14.33	11.32/13.65/12.16 12.37	10.42/12.86/11.63 11.64	9.86/12.30/11.00 11.05	9.67/12.22/10.86 10.91
W4 <sup>asym</sup> 128 and A8 <sup>asym</sup> 128								
RTN	30.59/36.56/27.07 31.41	24.11/29.43/22.74 25.43	15.38/18.57/15.69 16.55	13.22/16.32/13.91 14.49	11.13/13.97/12.30 12.47	10.34/12.82/11.55 11.57	9.98/12.15/10.96 11.03	9.57/11.86/10.58 10.67
GPTQ	30.47/36.19/26.40 31.02	23.35/27.96/21.94 24.42	14.92/17.57/15.12 15.87	12.48/15.60/13.46 13.85	10.87/13.34/11.91 12.04	10.20/12.45/11.28 11.31	9.62/11.88/10.74 10.75	9.39/11.55/10.41 10.45
ZQ-Local								9.37/11.70/10.49 10.52
ZQ-Global	29.85/34.52/26.10 30.16	22.70/27.72/21.64 24.02	14.96/17.55/15.09 15.86	12.64/15.40/13.47 13.84	10.93/13.43/11.95 12.10	10.18/12.68/11.42 11.42	9.74/12.02/10.83 10.86	9.39/11.53/10.42 10.45

Table E.21: BLOOM full results of Table 6.

Method	560m	1.1b	1.7b	3b	7.1b	176b
W4 <sup>sym</sup> full row and A8 <sup>sym</sup> 128						
RTN	25.32/46.98/27.12 33.14	23.87/68.29/25.97 39.38	16.99/31.15/19.51 22.55	14.69/25.22/17.30 19.07	12.07/20.86/14.84 15.92	8.34/14.05/11.24 11.21
GPTQ	24.00/44.47/25.66 31.37	24.14/66.95/26.17 39.09	16.38/29.64/18.79 21.61	14.10/24.19/16.67 18.32	11.77/20.22/14.48 15.49	8.20/13.82/11.07 11.03
ZQ-Local						8.30/14.01/11.20 11.17
ZQ-Global	23.92/44.23/25.69 31.28	22.53/57.71/23.51 34.58	16.25/29.72/18.74 21.57	14.12/24.26/16.74 18.38	11.78/20.30/14.53 15.53	8.24/13.82/11.10 11.05
W4 <sup>asym</sup> 128 and A8 <sup>sym</sup> 128						
RTN	23.84/44.94/25.79 31.53	18.65/51.54/21.21 30.46	16.18/30.03/18.70 21.64	14.04/24.32/16.77 18.38	23.05/48.33/23.69 31.69	8.87/15.68/11.72 12.09
GPTQ	23.22/43.24/25.01 30.49	18.25/48.89/20.74 29.29	16.00/29.44/18.41 21.29	13.77/23.68/16.35 17.93	11.54/19.76/14.27 15.19	8.13/13.69/11.01 10.95
ZQ-Local						8.20/13.87/11.08 11.05
ZQ-Global	23.12/43.22/25.03 30.45	18.19/48.96/20.72 29.29	15.75/28.81/18.30 20.95	13.73/23.65/16.39 17.92	11.57/19.85/14.32 15.25	8.17/13.76/11.03 10.99
W4 <sup>asym</sup> full row and A8 <sup>asym</sup> 128						
RTN	25.30/46.87/27.10 33.09	23.90/68.31/25.98 39.39	16.96/31.09/19.48 22.51	14.68/25.19/17.28 19.05	12.07/20.86/14.84 15.92	8.34/14.06/11.24 11.21
GPTQ	23.97/44.15/25.62 31.24	24.61/68.19/26.53 39.78	16.36/29.77/18.81 21.65	14.10/24.17/16.66 18.31	11.78/20.32/14.49 15.53	8.20/13.82/11.07 11.03
ZQ-Local						8.32/13.97/11.20 11.16
ZQ-Global	23.88/44.40/25.68 31.32	22.63/57.91/23.39 34.64	16.25/29.77/18.74 21.59	14.17/24.24/16.74 18.38	11.77/20.28/14.52 15.52	8.25/13.82/11.10 11.06
W4 <sup>asym</sup> 128 and A8 <sup>asym</sup> 128						
RTN	23.83/44.89/25.77 31.50	18.63/51.46/21.19 30.43	16.16/29.95/18.68 21.60	14.03/24.27/16.75 18.35	23.51/49.07/23.96 32.18	8.85/15.65/11.72 12.08
GPTQ	23.26/43.24/25.00 30.50	18.18/48.84/20.73 29.25	16.05/29.34/18.42 21.27	13.69/23.56/16.34 17.86	11.54/19.75/14.28 15.19	8.14/13.71/11.02 10.96
ZQ-Local						8.19/13.90/11.07 11.06
ZQ-Global	23.12/43.14/25.01 30.42	18.18/48.99/20.73 29.30	15.71/28.73/18.30 20.91	13.74/23.68/16.39 17.94	11.56/19.85/14.31 15.24	8.17/13.78/11.04 11.00

Table E.22: Full results of Table 6.

Block Size	1024	512	256	128	64	32
PPL	8.16/13.75/11.04	8.15/13.75/11.02	8.15/13.70/11.01	8.13/13.69/11.01	8.14/13.69/11.01	8.14/13.69/11.01

Table E.23: Results of applying LoRC on top of ZQ-Global for INT8 Activation.

model-size	precision	LoRC-dim	Learning Rate					Best
			0.0005	0.0001	5.00E-05	1.00E-05	5.00E-06	
125m	W4A8	0	4482.1	31.15	30.40	30.55	30.72	30.40
		8	5996.14	30.96	30.24	30.37	30.61	30.24
		16	3577.12	31.02	30.26	30.2	30.37	30.20
125m	W3A8	0	4283.28	41.03	40.93	55.74	86.34	40.93
		8	2396.92	37.25	36.65	37.85	39.06	36.65
		16	1787.74	36.66	36.55	37.46	38.21	36.55
125m	W2A8	0	3473.18	583.72	996.76	2480.69	3203.11	583.72
		8	3815.37	144.85	160.71	362.17	466.98	144.85
		16	3324.23	135.25	156.28	295.78	372.7	135.25
LoRC-dim			Learning Rate					best
350m	W4A8	0	25.65	24.38	24.34	24.55	24.75	24.34
		8	25.56	24.3	24.24	24.45	24.66	24.24
		16	25.45	24.39	24.21	24.39	24.63	24.21
350m	W3A8	0	30.59	28.45	28.94	31.51	32.39	28.45
		8	30.1	28.22	28.71	30.81	32.09	28.22
		16	30.64	28.02	28.50	30.62	31.69	28.02
350m	W2A8	0	97.40	177.43	257.61	668.19	722.19	97.4
		8	95.79	139.68	194.36	437.18	459.92	95.79
		16	106.51	137.81	172.93	400.91	421.59	106.51