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

Yujia Zhai,*¶ Chengquan Jiang,†¶ Leyuan Wang,† Xiaoying Jia,† Shang Zhang,‡
Zizhong Chen,* Xin Liu,†§ Yibo Zhu†
*University of California, Riverside
†ByteDance Ltd.
‡NVIDIA Corporation
§Correspondence to liuxin.ai@bytedance.com
¶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 subsequent 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

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

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

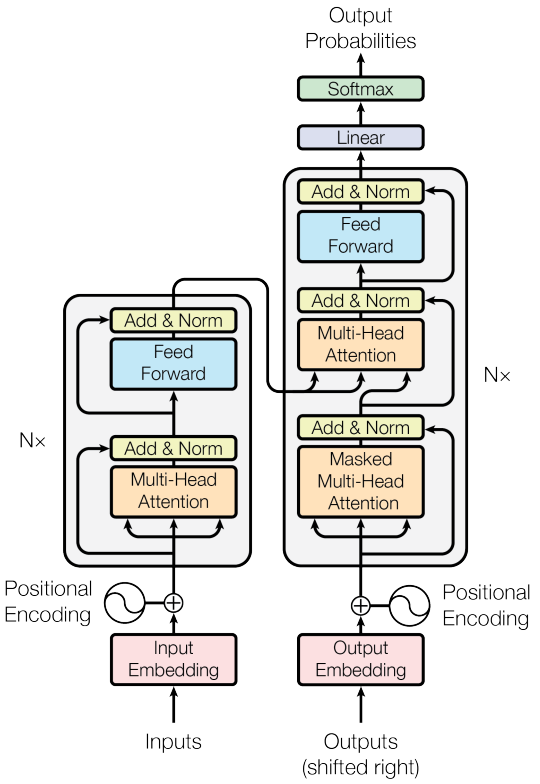


Fig. 1: The transformer architecture. [1]

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:

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

Whereas the formula of multi-head attention is: $Multihead(Q, K, V) = Concat(head_i, ..., head_h)$, here $head_i = Attention(Q_i, K_i, V_i)$.

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	≤ 512	no
TurboTransformer	yes	yes	no	partially
ByteTransformer	**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).

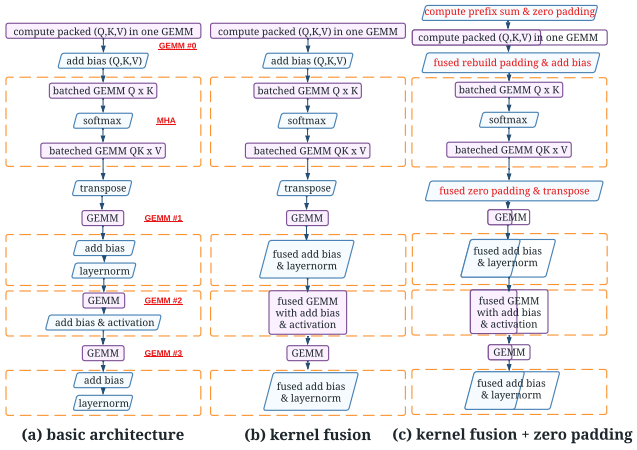
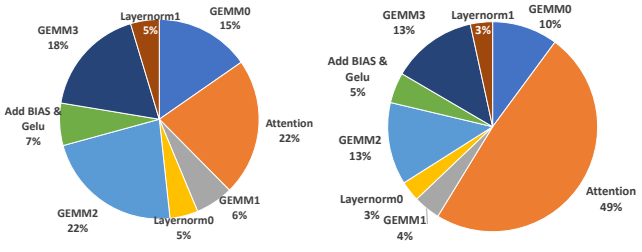


Fig. 2: BERT transformer architecture and optimizations.

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%.



(a) Sequence lengths 256 (b) Sequence lengths 1024

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` $\times$ `hidden_dim`) needs to first be added upon the input tensor (bias) and perform layer normalization. Here hidden dimension (`hidden_dim`) equals `head_num` $\times$ `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

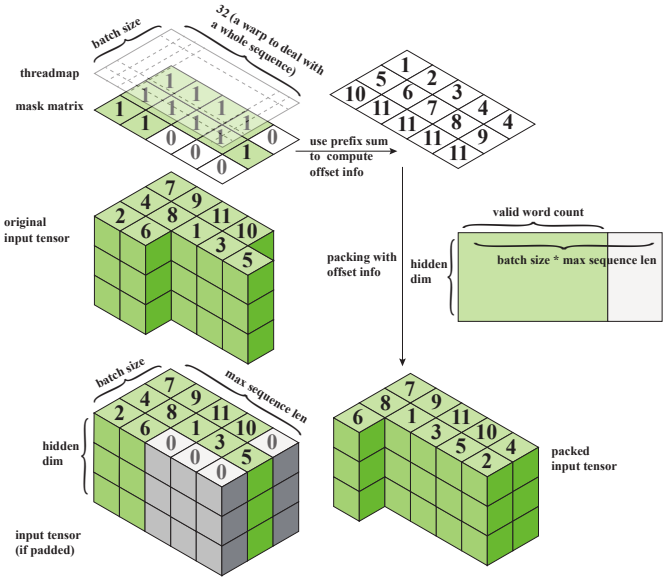


Fig. 4: The zero padding algorithm.

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` $\times$ `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 = α * maximum, m denotes `batch_size` $\cdot$ `max_seq_len`, k is denote hidden dimension `head_num` $\cdot$ `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`$/16 \times$ (`seq_len`$/16$). Each threadblock loads a chunk of Q (`split_seq_len` $\times$ `head_size`), K (`max_seq_len` $\times$ `head_size`) and V ((`head_size` $\times$

Algorithm III.1: Unpadded fused MHA for short sequences

```
/* define skew offset to avoid bank conflict */
#define SKEW_HALF 8
Shared memory:
__half s_kv [max_seq_len][size_per_head + SKEW_HALF];
__half s_query [split_seq_len][size_per_head +
  SKEW_HALF];
__half s_logits [max_seq_len][size_per_head +
  SKEW_HALF];
/* warps collaboratively fill s_query with adding bias fused */
Load __half2 q_bias
for seq_id = warp_id : warp_num : split_seq_len do
    query = Q[batch_seq_offset + seq_id +
      thread_offset];
    offset = seq_id*(head_size+SKEW_HALF)+(lane_id*2);
    (__half2 *)s_query[offset] = fast_add(query,
      k_bias);
/* warps collaboratively fill s_kv with adding bias fused */
Load __half2 k_bias
for seq_id = warp_id : warp_num : batch_seq_len do
    key = K[batch_seq_offset + seq_id +
      thread_offset];
    offset = seq_id*(head_size+SKEW_HALF)+(lane_id*2);
    (__half2 *)s_kv[offset] = fast_add(key, k_bias);
/* compute Q*K using WMMA */
Clear wmma fragment QK to zero
for k_id = 0 : head_size / 16 do
    Load 16x16 wmma fragments of Q
    Load 16x16 wmma fragments of K
    Update QK = Q * K + QK using wmma::mma_sync
Store fragment QK to s_logits using wmma::store_matrix_sync
/* Compute softmax */
for seq_id = warp_id : warp_num : batch_seq_len do
    float logits[max_seq_len];
    each thread loads a whole sequence to fill local registers
    /* 1st round of reduction with register-level data re-use*/
    compute max_val in local registers
    /* register-level data re-use*/
    compute P = exp(P − max) and update local registers
    /* 2st round of reduction with register-level data re-use*/
    compute sum_val in local registers
    /* register-level data re-use*/
    compute P = P/sum_val and stream to s_logits
/* warps collaboratively fill s_kv with adding bias fused */
Load __half2 v_bias
for seq_id = warp_id : warp_num : batch_seq_len do
    value = V[batch_seq_offset + seq_id +
      thread_offset];
    offset = seq_id*(head_size+SKEW_HALF)+(lane_id*2);
    (__half2 *)s_kv[offset] = fast_add(value, v_bias);
/* Similar to Q * K so omitting the details here */
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

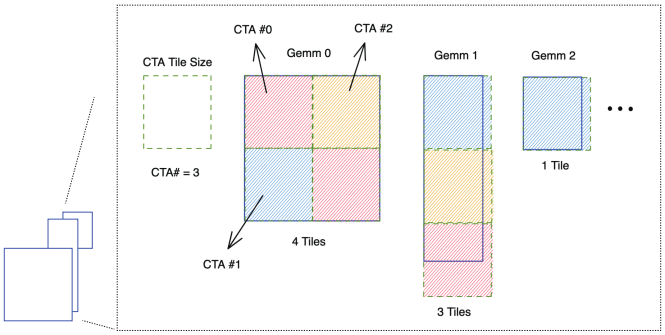


Fig. 5: Grouped GEMM demonstration.

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.

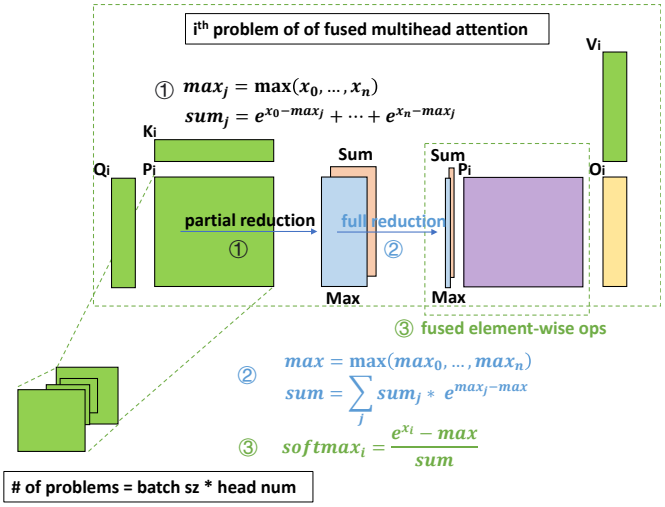


Fig. 6: Grouped-GEMM-based FMHA. The prototype of our fused MHA has been upstreamed to and released with CUTLASS 2.10. Source codes are available at [32].

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` $\times$ `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].

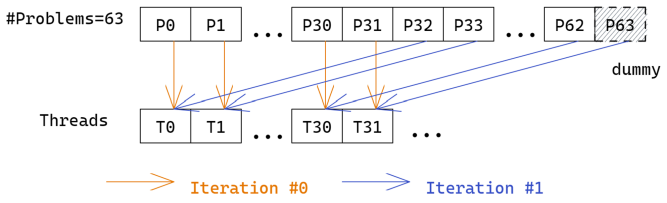


Fig. 7: Warp prefetching for grouped GEMM.

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 M_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

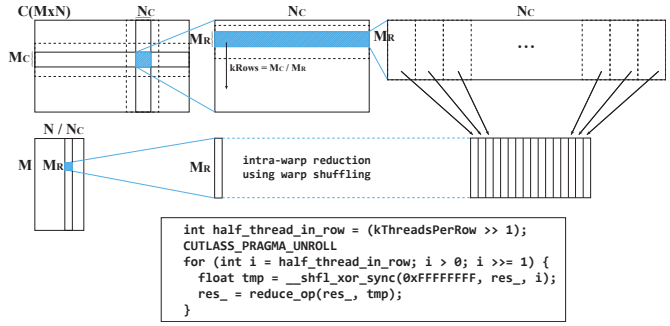


Fig. 8: Fused softmax reduction in grouped GEMM epilogue.

the threadmap is arranged as 8×16, where each thread holds a 128-bit register tile in each step. After the intra-thread reduction, the $M_R \times N_C$ (8×128) sub-matrix is reduced to 8×16, 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` $\times$ `seq_len` while the full reduction just reduces a `seq_len` $\times$ `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 $\frac{e^{x_{ij}} - max}{sum}$ 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

```
Register Tiles:
WarpLoadedFragmentA warp_loaded_frag_A[2];
WarpLoadedFragmentB warp_loaded_frag_B[2];
WarpLoadedFragmentNormSum warp_loaded_frag_norm_sum;
Shared memory: (kStages + 1) shared-memory tiles for A and B
/* prologue */
Load k-invariant fused softmax tile to warp_loaded_frag_norm_sum
Prefetch kStages - 1 tiles of A to shared memory using cp.async
Prefetch kStages - 1 tiles of B to shared memory using cp.async
Prefetch a tile of A from shared memory to warp_loaded_frag_A[0]
Prefetch a tile of B from shared memory to warp_loaded_frag_B[0]
/* fused element-wise operation */
/* A = exp(A-max)/sum */
elementwise_transform(
 warp_loaded_frag_A[0],
 warp_loaded_frag_norm_sum);
/* mainloop */
for k to -kStages + 1 do
    /* Computes a warp-level GEMM */
    /* with pipelined load during iterations */
    for warp_mma_k = 0 to kWarpGemmIterations - 1 do
        Prefetch warp_loaded_frag_A[(warp_mma_k + 1) % 2]
        Prefetch warp_loaded_frag_B[(warp_mma_k + 1) % 2]
        /* fused element-wise transform */
        elementwise_transform(
         warp_loaded_frag_A[(warp_mma_k + 1) % 2],
         warp_loaded_frag_norm_sum);
        /* Computes a warp-level GEMM*/
        /* on data loaded in previous iteration */
        warp_mma(
         accum,
         warp_loaded_frag_A[warp_mma_k % 2],
         warp_loaded_frag_B[warp_mma_k % 2],
         accum);
        Prefetch a tile of A to shared memory using cp.async
        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.

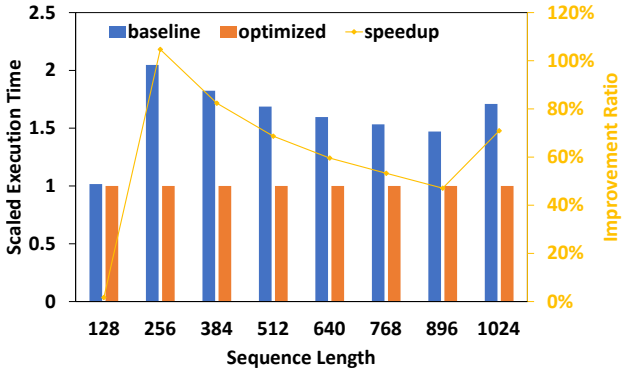


Fig. 9: Kernel fusion for add-bias and layernorm on a $(\texttt{batch\_size} \cdot \texttt{seq\_len}) \times \texttt{hidden\_dim}$ tensor. Here we profile for 16 batches with the hidden dimension fixed to 768 under the standard BERT configuration.

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.

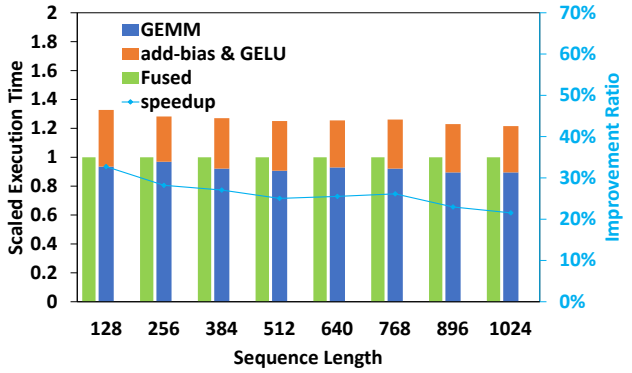


Fig. 10: Kernel fusion for GEMM, add-bias, and GELU. The shape of output tensor is $(\texttt{batch\_size} \cdot \texttt{seq\_len}) \times (\texttt{scale} \cdot \texttt{hidden\_dim})$. Here we profile for 16 batches with the hidden dimension and the scale factor fixed to 768 and 4 under the standard BERT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

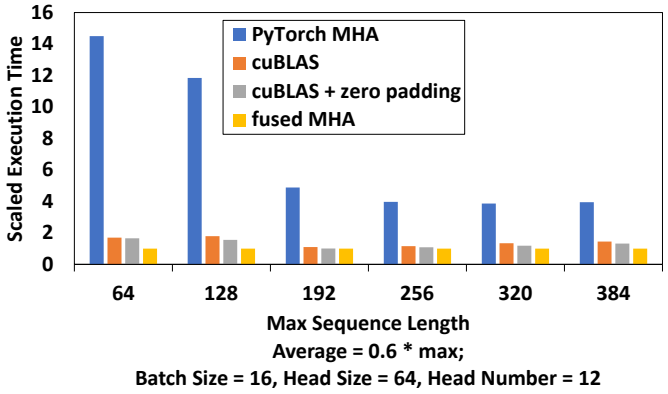


Fig. 11: Fused MHA for short sequences.

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.

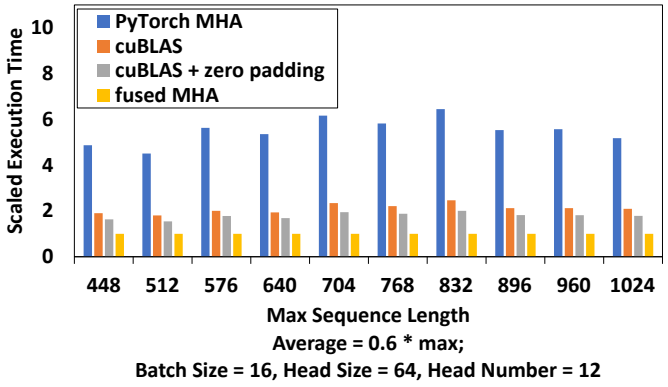


Fig. 12: Fused MHA for long sequences.

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

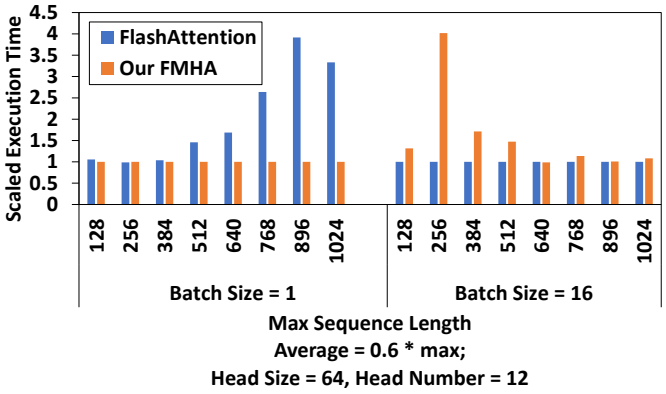


Fig. 13: Comparisons of our FMHA with FlashAttention.

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.

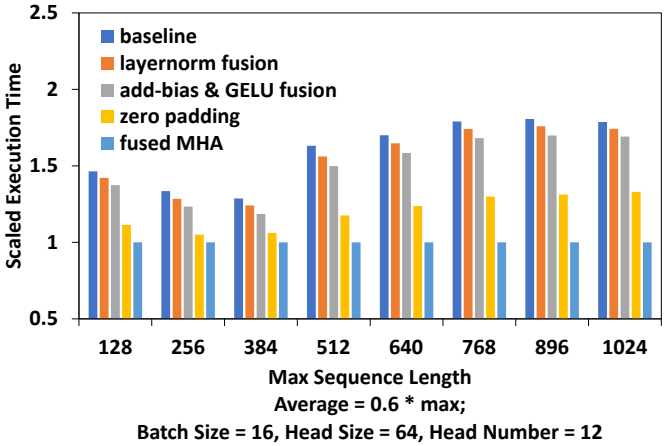


Fig. 14: Single-layer BERT transformer with step-wise optimizations. Each variant includes all previous optimizations.

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.

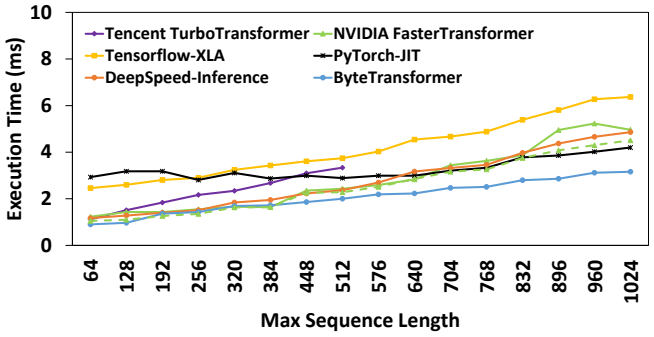
Sequence Length	E.T. (ms)	ByteTransformer (ms)	Speedup
256	0.25	0.07	3.57×
1024	1.04	0.09	11.56×

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.

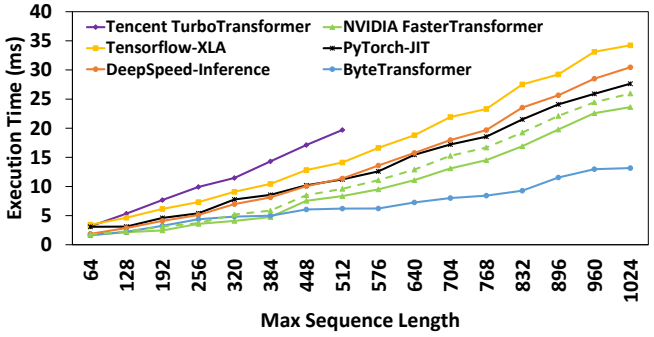
E. Benchmarking end-to-end performance of BERT

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, Micorsoft 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.

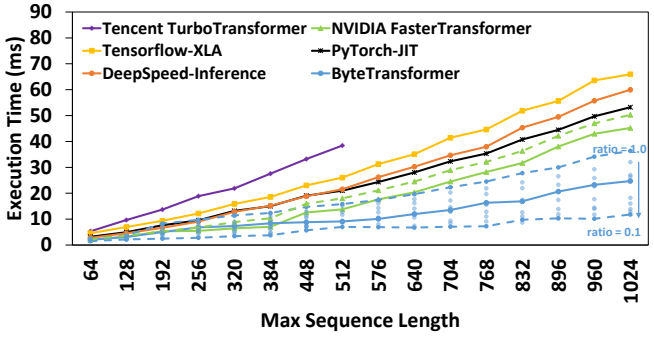
Compared with popular DL frameworks PyTorch, TensorFlow, and Microsoft DeepSpeed-Inference, our ByteTransformer achieves 87%, 131%, and 74% faster end-to-end performance on average. When benchmarking Tencent TurboTransformer, we turn on its `SmartBatch` mode to reach optimal batching performance. Since 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



(a) Batch size = 1



(b) Batch size = 8



(c) Batch size = 16

Fig. 15: End-to-end benchmark for standard BERT transformer, head size = 64, head number = 12, layer = 12, average sequence length = 0.6 * 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. TurboTransfomer 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.

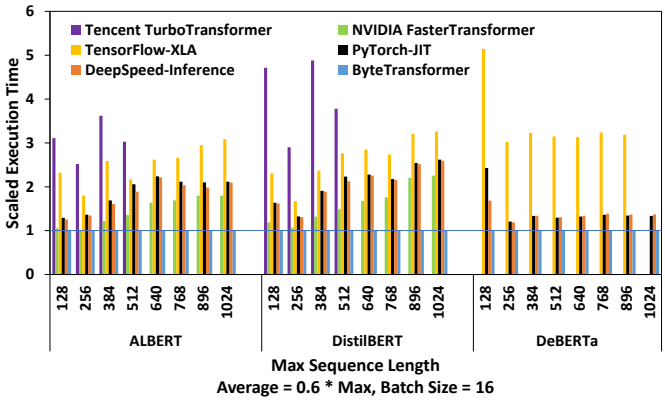


Fig. 16: End-to-end benchmark for other BERT-like models.

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. Lyashevsky, "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/bertQKVToContextPluginl, 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, Retrieved in 2022, online.

[33] NVIDIA, 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. Debut, 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

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*
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× 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

*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[1] 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, 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× 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].

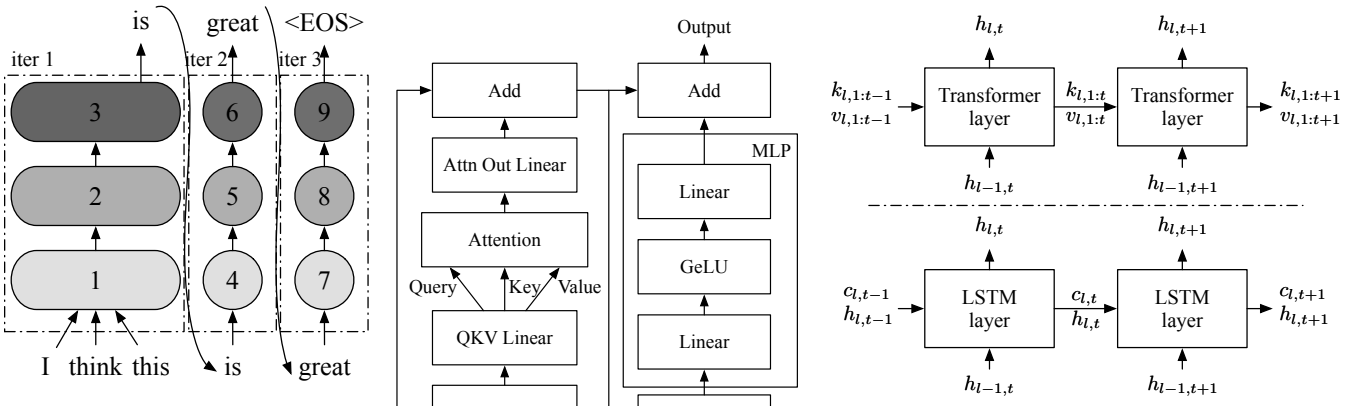
2 Background

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

[1] 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.



(a) A computation graph representing an inference procedure using a GPT model. The graph does not depict layers other than Transformer layers (e.g., embedding) for simplicity.

(b) A Transformer layer used in GPT.

(c) Internal state usage of Transformer. h, k, v, and c refer to layer input/output, Attention key, Attention value, and LSTM internal memory, respectively. l denotes layer index and t denotes token index.

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,[2] 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.

[2]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 $v_{l,t}$.[3] 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

[3]$k_{l,1:t-1}$ 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 $v_{l,t}$.

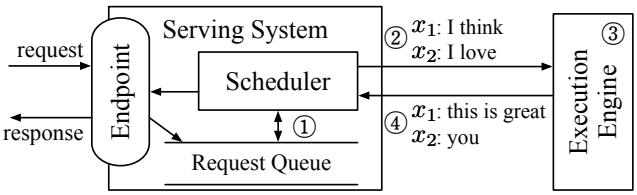


Figure 2: Overall workflow of serving a generative language model with existing systems.

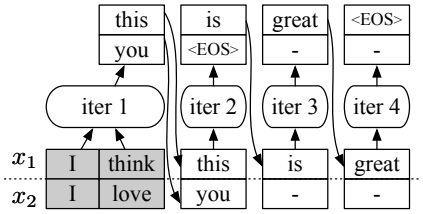


Figure 3: An illustration for a case where the requests have the same input length but some requests finish earlier than others. Shaded tokens represent input tokens. "-" denotes inputs and outputs of extra computation imposed by the scheduling.

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 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].

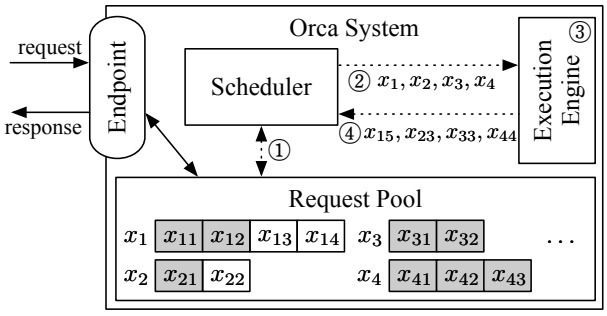


Figure 4: System overview of ORCA. Interactions between components represented as dotted lines indicate that the interaction takes place at every iteration of the execution engine. x_{ij} is the j-th token of the i-th request. Shaded tokens represent input tokens received from the clients, while unshaded 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,

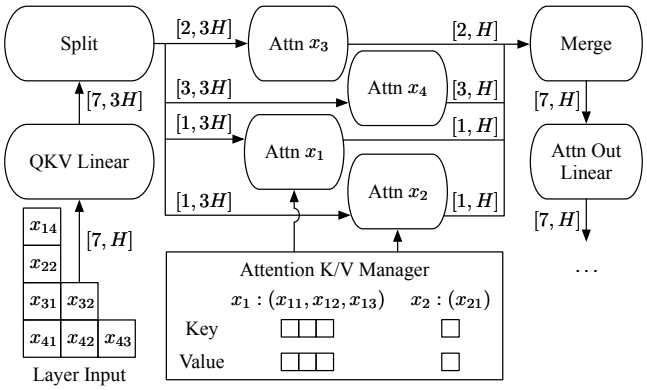


Figure 5: An illustration of ORCA execution engine running a Transformer layer on a batch of requests with selective batching. We only depict the QKV Linear, Attention, and Attention Out Linear operations for simplicity.

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[4] 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. 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, LayerNorm, 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,

[4] We use the first case as a driving example, but the argument can be similarly applied to the other two cases.

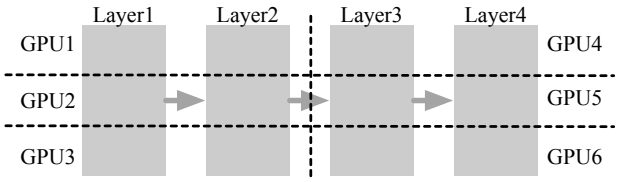


Figure 6: An example of intra- and inter- layer parallelism. A vertical dotted line indicates partitioning between layers and a horizontal line indicates partitioning within a layer.

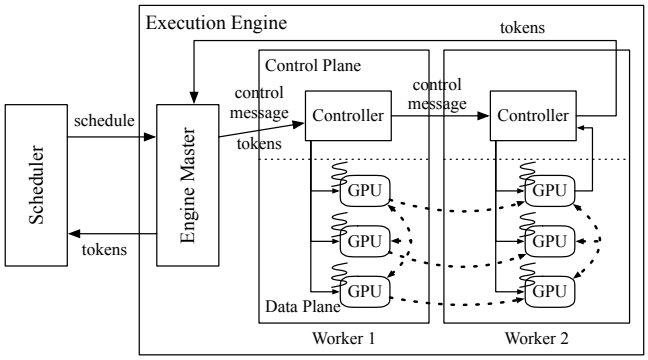


Figure 7: An illustration of the distributed architecture of ORCA's execution engine using the parallelization configuration shown in Figure 6. For example, the first inter-layer partition (Layer1 and Layer2) in Figure 6 is assigned to Worker1, while the second partition is assigned to Worker2.

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[5] 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-

[5]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*.

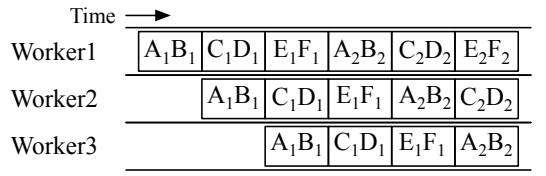
Algorithm 1: ORCA scheduling algorithm

```
   Params: n_workers: number of workers, max_bs:
           max batch size, n_slots: number of K/V slots
1  n_scheduled ← 0
2  n_rsrv ← 0
3  while true do
4      batch, n_rsrv ← Select(request_pool, n_rsrv)
5      schedule engine to run one iteration of
       the model for the batch
6      foreach req in batch do
7          req.state ← RUNNING
8      n_scheduled ← n_scheduled + 1
9      if n_scheduled = n_workers then
10         wait for return of a scheduled batch
11         foreach req in the returned batch do
12             req.state ← INCREMENT
13             if finished(req) then
14                 n_rsrv ← n_rsrv − req.max_tokens
15         n_scheduled ← n_scheduled − 1
16
17 def Select(pool, n_rsrv):
18     batch ← {}
19     pool ← {req ∈ pool | req.state ≠ RUNNING}
20     SortByArrivalTime(pool)
21     foreach req in pool do
22         if batch.size() = max_bs then break
23         if req.state = INITIATION then
24             new_n_rsrv ← n_rsrv + req.max_tokens
25             if new_n_rsrv > n_slots then break
26             n_rsrv ← new_n_rsrv
27         batch ← batch ⋃ {req}
28     return batch, n_rsrv
```

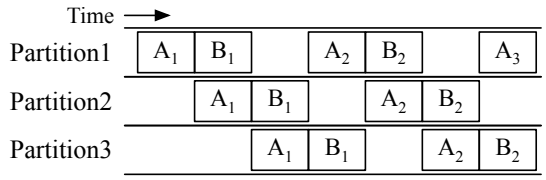
When the scheduler considers a request in the initiation phase, meaning that the request has never been scheduled yet, the scheduler uses the request's *max_tokens*[6] attribute to reserve *max_tokens* slots of GPU memory for storing the keys and values in advance (lines 23-26). The scheduler determines 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

[6] The *max_tokens* attribute is a per-request option, meaning the maximum number of tokens that a request can have after processing.



(a) ORCA execution pipeline.



(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 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.

# 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.

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].

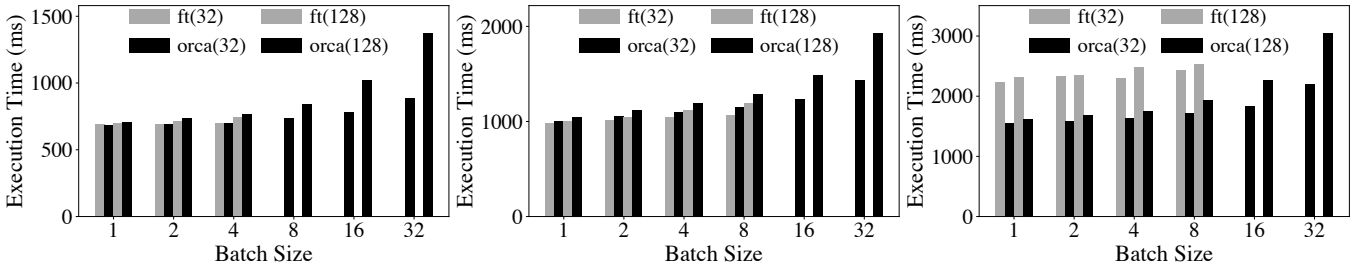
The second scenario tests the end-to-end performance of ORCA by emulating a workload. We synthesize a trace of client requests because there is no publicly-available request trace for generative language models. Each request in the synthesized trace is randomly generated by sampling the number of input 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.

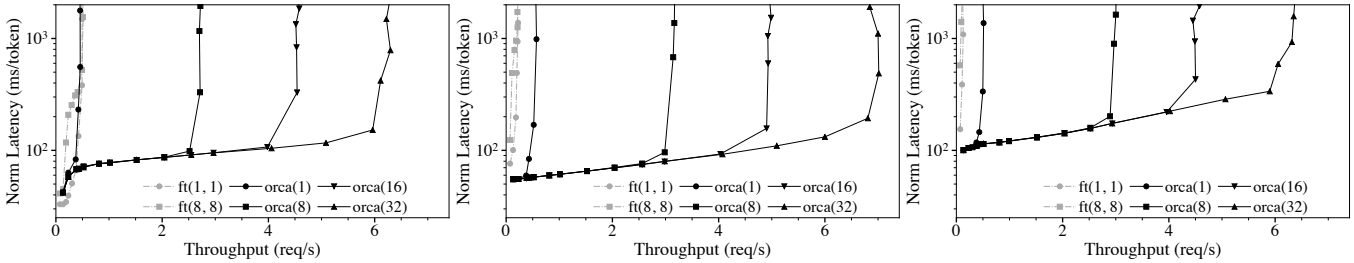
Figure 9 shows the performance of FasterTransformer and the ORCA engine for processing a batch composed of the same requests. In Figure 9a, the ORCA engine shows a similar (or slightly worse) performance compared to FasterTransformer across all configurations. This is because ORCA does not apply batching 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



(a) 13B model, 1 GPU. (b) 101B model, 8 GPUs. (c) 175B model, 16 GPUs.

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 incurs out of memory error are represented as missing entries (e.g., ft(32) for the 101B model with a batch size of 16).



(a) 101B model, 8 GPU. (b) 175B model, 16 GPUs. (c) 341B model, 32 GPUs.

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.

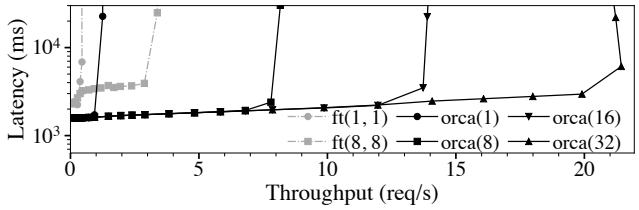
Next, we go one step further and experiment with the 175B model, which splits the layers into two inter-layer partitions. In this case, for better comparison, we disable pipelined execution of the inter-layer partitions for both systems. For FasterTransformer, we set the size of a microbatch to be equal 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

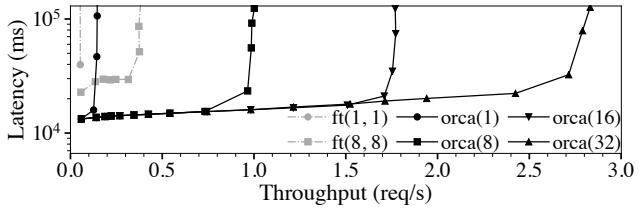
Now we assess the end-to-end performance of ORCA by measuring the latency and throughput with the synthesized request trace under varying load. When synthesizing the trace, we sample each request's number of input tokens from $U(32,512)$, a uniform distribution ranging from 32 to 512 (inclusive). The *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



(a) (# in, # gen) = (32, 32)



(b) (# in, # gen) = (256, 256)

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× 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 Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, Giri Anantharaman, Xian Li, Shuohui 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 Hebgen 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, 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. *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 Pellat, 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 Millican, 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, Lintao 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, Çaglar Gülçehre, 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, Saffron 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, Lorrayne 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. Mesh-TensorFlow: 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, Shuohui 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.

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× and 6.4×, 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× and 6.4×, 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.

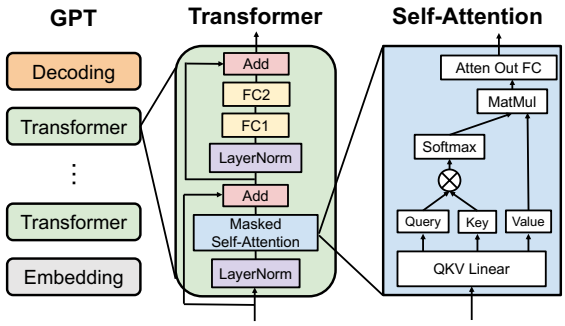


Figure 1. GPT model architecture.

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

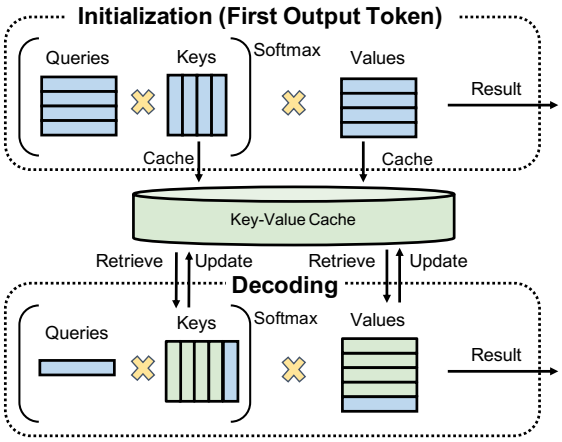


Figure 2. Example usage of KV cache in two phases. The prompt contains four tokens. Blue grids represent newly computed tensors in its iteration, while green grids represent reused tensors from the key-value cache.

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.

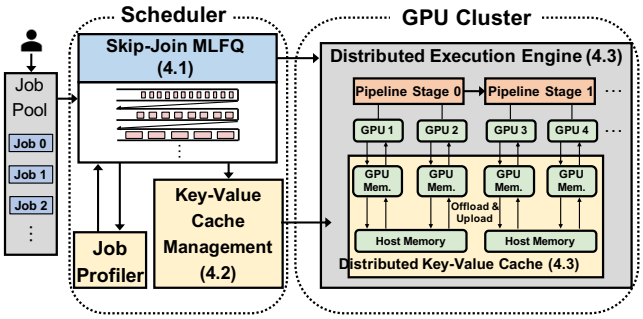


Figure 3. FastServe architecture.

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.

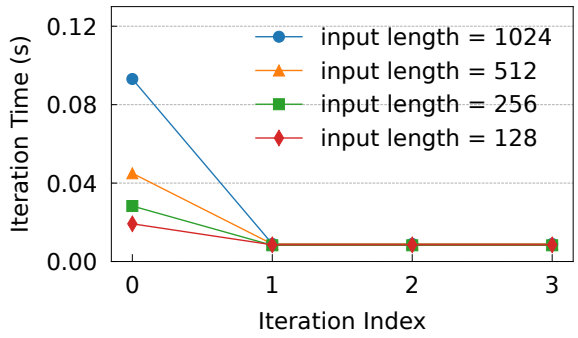


Figure 4. The execution time of the first four iterations (i.e., first four output tokens) with different input sequence length.

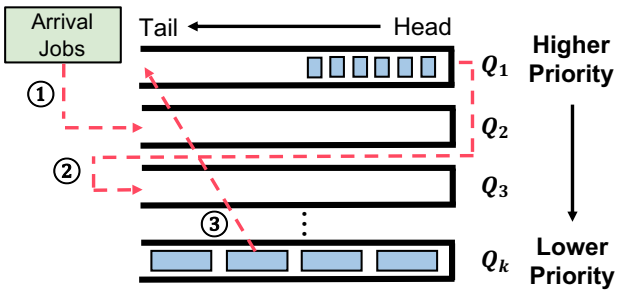


Figure 5. Skip-join MLFQ with starvation prevention.

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 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 ❶ 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 ❷ 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 ❸ 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

Input: Queues $Q_1, Q_2, ..., Q_k$, newly arrived jobs J_{in}, preempted jobs J_{pre}, and profiling information P
Output: Jobs to be executed J_{out}
procedure SkipJoinMLFQScheduler
    Initialization: $J_{out} \leftarrow \emptyset$.
    // Process newly arrival jobs.
    for $job \in J_{in}$ **do**
        $nextIterTime \leftarrow P.getNextIterTime(job)$
        $p_{job} \leftarrow getHighestPriority(nextIterTime)$
        $Q_{p_{job}}.push(job)$
    // Process preempted jobs.
    for $job \in J_{pre}$ **do**
        $job.outputNewGeneratedToken()$
        $p_{job} \leftarrow job.getCurrentPriority()$
        if $job.isFinished()$ **then**
            $Q_{p_{job}}.pop(job)$
            continue
        if $job.needDemotion()$ **then**
            $nextIterTime' \leftarrow P.getNextIterTime(job)$
            $p'_{job} \leftarrow getDemotionPriority(p_{job}, nextIterTime')$
            $r.demoteTo(Q_{p'_{job}})$
    // Promote starved jobs.
    for $q \in \{Q_2, Q_3, ..., Q_k\}$ **do**
        for $job \in q$ **do**
            if $job.needPromotion()$ **then**
                $job.promoteTo(Q_1)$
                $job.resetStarveTimer()$
    // Schedule jobs to be executed.
    for $q \in \{Q_1, Q_2, ..., Q_k\}$ **do**
        for $job \in q$ **do**
            if $job.isReady()$ **and** $|J_{out}| < MaxBatchSize$ **then**
                $J_{out}.push(job)$

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.

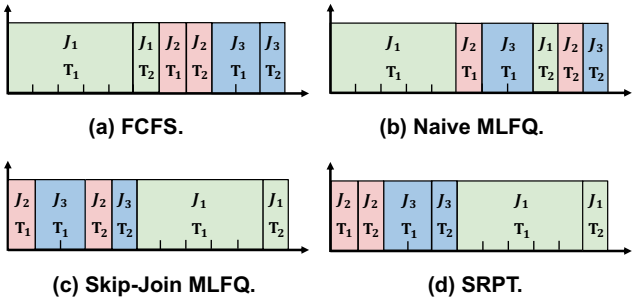


Figure 6. Execution timeline of three jobs with different scheduling algorithms. The job index J_i and generated token index T_i are marked in each iteration block.

4.2 Proactive Key-Value Cache Management

The skip-join MLFQ scheduler provides iteration-level preemption 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× 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

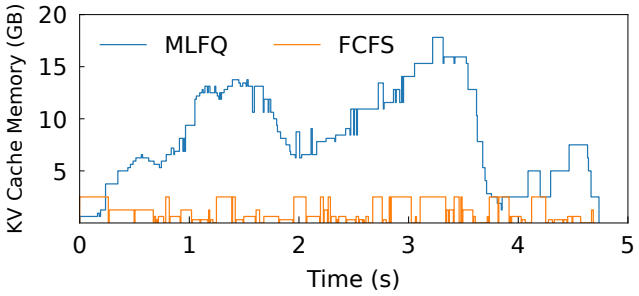


Figure 7. The key-value cache memory consumption for GPT-3 2.7B under different schedulers. The workload follows a Gamma Process with rate=64 and CV=4. The maximum output length is set to 20 to avoid GPU out of memory.

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 $T_{promote}(i)$. 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:

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

where $i.priority$ is the priority of job i, and $quantum(k)$ 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:

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

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 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.

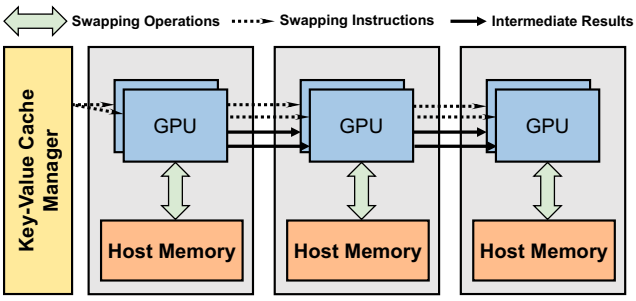


Figure 8. Overlapping key-value cache offloading with intermediate result transmission to reduce offloading overhead.

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 is broadly adopted in many open-source big data benchmarks [13, 17, 27, 55]. The Zipf distribution is parameterized by one parameter, θ, which controls the skewness of the distribution. The larger θ 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.

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.

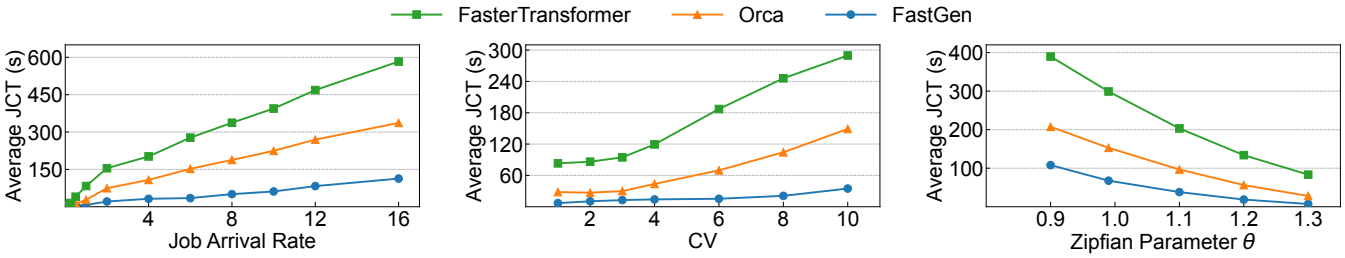
Metrics. Since the user-perceived latency is a critical measurement for interactive applications like ChatGPT, which FastServe targets at, we use job completion time (JCT) as the major evaluation metric. Due to limited space, we show average JCT for most experiments, and report both average and tail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



(a) Average JCT vs. load. (b) Average JCT vs. burstiness. (c) Average JCT vs. skewness.

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 (≤ 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 reposefully. 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 θ, 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 θ 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 θ 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 θ decreases. This is because we bound the maximum input and output lengths. As a result, the workloads with smaller θ (i.e., balanced job lengths) have more tokens to process.

6.3 Benefits of Design Choices

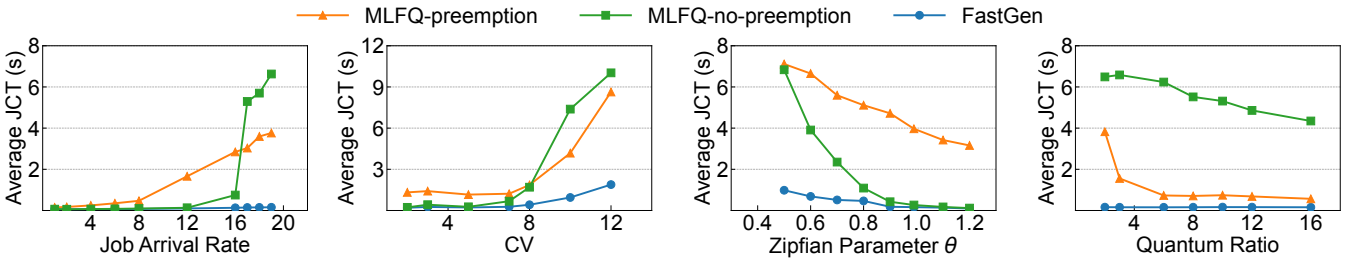
In this subsection, we study the effectiveness of FastServe's main techniques: skip-join MLFQ scheduler and proactive key-value cache management. Due to a limited budget, we use one A100 GPU to run GPT-3 2.7B in the experiments.

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.

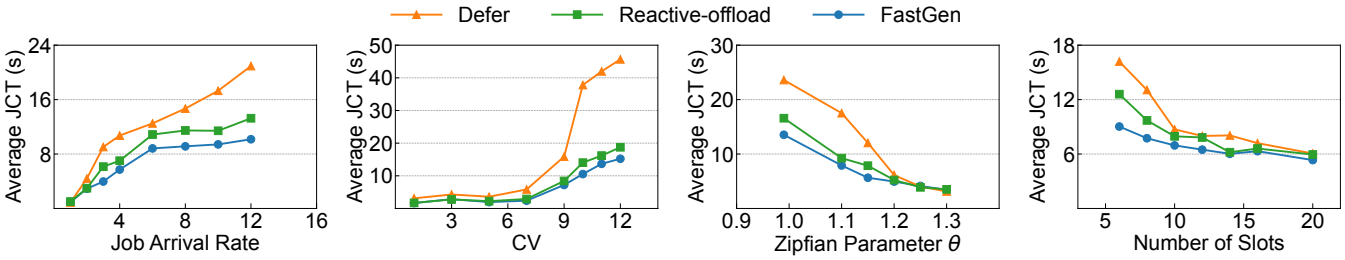
Similar to previous experiments, we vary the rate, CV, and Zipfian parameter of the workload. In addition, to evaluate the sensitivity of MLFQ to the quantum settings, we vary the *quantum ratio* (§4.1) to see the impact on performance. The results are summarized as follows.

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



(a) Ave. JCT vs. load. (b) Ave. JCT vs. burstiness. (c) Ave. JCT vs. skewness. (d) Ave. JCT vs. quantum ratio.

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



(a) Ave. JCT vs. load. (b) Ave. JCT vs. burstiness. (c) Ave. JCT vs. skewness. (d) Ave. JCT vs. cache size.

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 θ decreases. This is because a small θ 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.

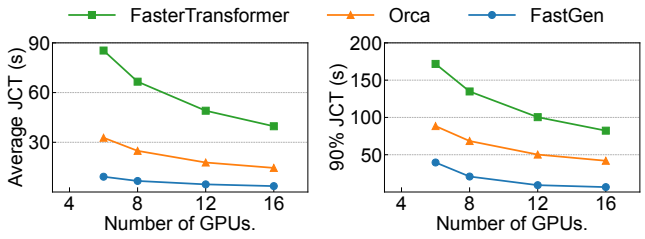
In Figure 10(d), it is worth noting that increasing the quantum ratio has little impact on the performance of FastServe, but it reduces the JCT of the two baseline MLFQ schedulers. This demonstrates that FastServe is not sensitive to the quantum settings, making the life of the system administrator much 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



(a) Average JCT. (b) Tail JCT.

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 θ 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, FastServe supports iter-job pipeline parallelism in its scheduler, and benefits from memory locality through its distributed key-value cache management. The results show that FastServe 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× and 6.4× 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 Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. *arXiv* (2022).

[25] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, 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 μsecond-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 μsecond-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, Yinmin 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. (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, Pierric 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*.

FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

Tri Dao[†], Daniel Y. Fu[†], Stefano Ermon[†], Atri Rudra[‡], and Christopher Ré[†]

[†]Department of Computer Science, Stanford University
[‡]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

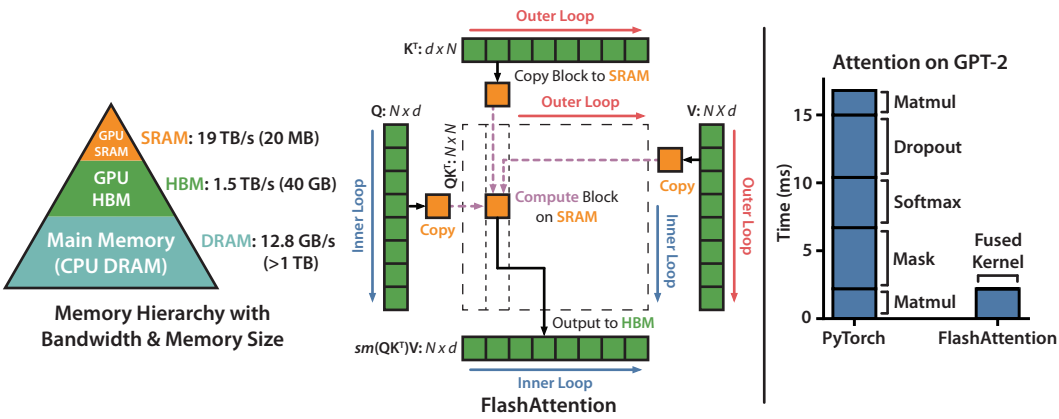


Figure 1: **Left:** FlashAttention uses tiling to prevent materialization of the large $N \times N$ attention matrix (dotted box) on (relatively) slow GPU HBM. In the outer loop (red arrows), FlashAttention loops through blocks of the **K** and **V** matrices and loads them to fast on-chip SRAM. In each block, FlashAttention loops over blocks of **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 9× 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 2-4× 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.[1]

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].

[1] 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 $\mathbf{O} \in \mathbb{R}^{N \times d}$:

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

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 $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, 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:

$$m(x) := \max_i \; x_i, \quad f(x) := \begin{bmatrix} e^{x_1 - m(x)} & \dots & e^{x_B - m(x)} \end{bmatrix}, \quad \ell(x) := \sum_i f(x)_i, \quad \mathrm{softmax}(x) := \frac{f(x)}{\ell(x)}.$$

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

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

$$\ell(x) = \ell(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = e^{m(x^{(1)})-m(x)} \ell(x^{(1)}) + e^{m(x^{(2)})-m(x)} \ell(x^{(2)}), \quad \text{softmax}(x) = \frac{f(x)}{\ell(x)}.$$

Therefore if we keep track of some extra statistics $(m(x), \ell(x))$, we can compute softmax one block at a time.[2] 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, ℓ), 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 $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ 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.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ 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: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **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* $O(N^2 d)$ *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

[2] This style of aggregation is called *algebraic aggregation* [33].

Attention	Standard	FlashAttention
GFLOPs	66.6	75.2
HBM R/W (GB)	40.3	4.4
Runtime (ms)	41.7	7.3

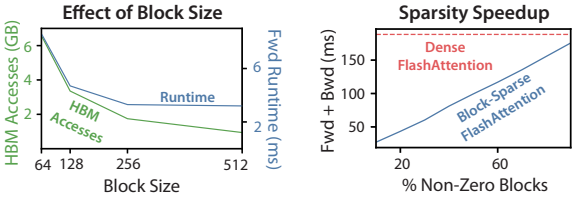


Figure 2: **Left**: Forward + backward runtime of standard attention and FlashAttention for GPT-2 medium (seq. length 1024, head dim. 64, 16 heads, batch size 64) on A100 GPU. HBM access is the primary factor affecting runtime. **Middle**: Forward runtime of FlashAttention (seq. length 1024, head dim. 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 $d \le M \le Nd$. 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 $d \le M \le Nd$. 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:

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

where $(\mathbf{S} \odot \mathbb{1}_{\tilde{\mathbf{M}}})_{kl} = \mathbf{S}_{kl}$ if $\tilde{\mathbf{M}}_{kl} = 1$ and $-\infty$ if $\mathbf{M}_{kl} = 0$. 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 $d \le M \le Nd$. Block-sparse* FlashAttention *(Algorithm 5) requires $\Theta(Nd + N^2d^2M^{-1}s)$ 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× 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× over HuggingFace [87] and 1.8× over Megatron [77] over standard Transformers. FlashAttention speeds up the long-range arena (LRA) benchmark 2.4×.
- **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× 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×A100 GPUs.

BERT Implementation	Training time (minutes)
Nvidia MLPerf 1.1 [58]	20.0 ± 1.5
FlashAttention (ours)	**17.4** ± 1.4

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× end-to-end speedup compared to Huggingface and 1.7× 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× speed up compared to Huggingface implementation and up to 1.7× compared to Megatron-LM. Training time reported on 8×A100s GPUs.

Model implementations	OpenWebText (ppl)	Training time (speedup)
GPT-2 small - Huggingface [87]	18.2	9.5 days (1.0×)
GPT-2 small - Megatron-LM [77]	18.2	4.7 days (2.0×)
GPT-2 small - FlashAttention	18.2	**2.7 days (3.5×)**
GPT-2 medium - Huggingface [87]	14.2	21.0 days (1.0×)
GPT-2 medium - Megatron-LM [77]	14.3	11.5 days (1.8×)
GPT-2 medium - FlashAttention	14.3	**6.9 days (3.0×)**

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].[3] Table 3 shows that FlashAttention achieves up 2.4× 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×
Block-sparse FlashAttention	37.0	63.0	81.3	43.6	73.3	59.6	**2.8×**
Linformer [84]	35.6	55.9	77.7	37.8	67.6	54.9	2.5×
Linear Attention [50]	38.8	63.2	80.7	42.6	72.5	59.6	2.3×
Performer [12]	36.8	63.6	82.2	42.1	69.9	58.9	1.8×
Local Attention [80]	36.1	60.2	76.7	40.6	66.6	56.0	1.7×
Reformer [51]	36.5	63.8	78.5	39.6	69.4	57.6	1.3×
Smyrf [19]	36.1	64.1	79.0	39.6	70.5	57.9	1.7×

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× 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× larger context length compared to Megatron-LM, is still 30% faster while achieving 0.7 better perplexity. Training time on 8×A100 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×)
GPT-2 small - FlashAttention	1k	18.2	**2.7 days (1.7×)**
GPT-2 small - FlashAttention	2k	17.6	3.0 days (1.6×)
GPT-2 small - FlashAttention	4k	**17.5**	3.6 days (1.3×)

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

[3] 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].

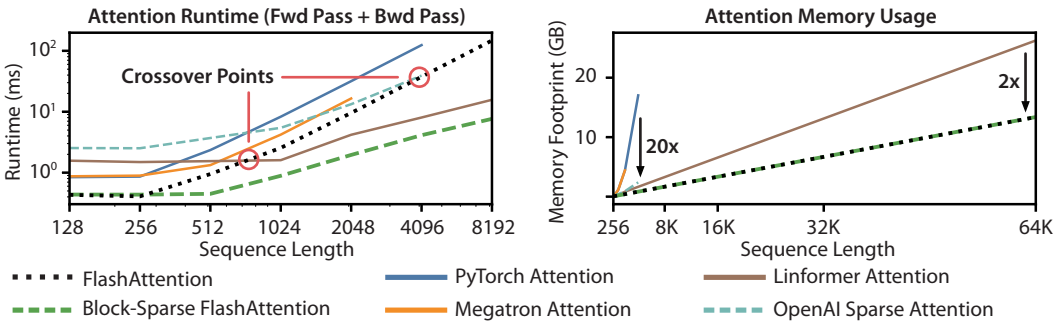


Figure 3: **Left:** runtime of forward pass + backward pass. **Right:** attention memory usage.

European Court of Human Rights, each of which is mapped to articles of the Convention of Human Rights that were allegedly violaged. 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 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	**57.1**
ECtHR [6]	72.2	74.3	77.1	78.6	**80.7**	79.2

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

Model	Path-X	Path-256
Transformer	✗	✗
Linformer [84]	✗	✗
Linear Attention [50]	✗	✗
Performer [12]	✗	✗
Local Attention [80]	✗	✗
Reformer [51]	✗	✗
SMYRF [19]	✗	✗
FlashAttention	**61.4**	✗
Block-sparse FlashAttention	56.0	**63.1**

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[4] 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.

[4]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× 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× 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× 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 Likhosherstov, 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, Łukasz 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 Likhosherstov, 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 Bittorf, 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, 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.

[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, Pierric 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 has 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 $\mathbf{O} \in \mathbb{R}^{N \times d}$:

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

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:

$$L_i = \sum_j e^{q_i^T k_j}. \tag{1}$$

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

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

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 ϕ, 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:

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

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

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

where $\circ$ denotes pointwise multiplication.

Define

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

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

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

Similarly,

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

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 $\mathbf{O} \in \mathbb{R}^{N \times d}$:

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

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 ℓ 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_{drop}.

Initialize the pseudo-random number generator state $\mathcal{R}$ and save to HBM.
Set block sizes $B_c = \left\lceil \frac{M}{4d} \right\rceil, B_r = \min\left(\left\lceil \frac{M}{4d} \right\rceil, d\right)$.
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.
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.
Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ 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.
for $1 \le j \le T_c$ **do**
  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
  for $1 \le i \le T_r$ **do**
    Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
    On chip, compute $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
    On chip, compute $\mathbf{S}_{ij}^{\text{masked}} = \text{MASK}(\mathbf{S}_{ij})$.
    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), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
    On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
    On chip, compute $\tilde{\mathbf{P}}_{ij}^{\text{dropped}} = \text{dropout}(\tilde{\mathbf{P}}_{ij}, p_{\text{drop}})$.
    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.
    Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
  end for
end for
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.

Load $\mathbf{P}, \mathbf{dO}$ by blocks from HBM, compute $\mathbf{dV} = \mathbf{P}^\top \mathbf{dO} \in \mathbb{R}^{N \times d}$, write $\mathbf{dV}$ to HBM.
Load $\mathbf{dO}, \mathbf{V}$ by blocks from HBM, compute $\mathbf{dP} = \mathbf{dO}\mathbf{V}^\top \in \mathbb{R}^{N \times N}$, write $\mathbf{dP}$ to HBM.
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.
Load $\mathbf{dS}$ and $\mathbf{K}$ by blocks from HBM, compute $\mathbf{dQ} = \mathbf{dSK}$, write $\mathbf{dQ}$ to HBM.
Load $\mathbf{dS}$ and $\mathbf{Q}$ by blocks from HBM, compute $\mathbf{dK} = \mathbf{dS}^\top \mathbf{Q}$, write $\mathbf{dK}$ to HBM.
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 $D_i = P_{i:}^\top dP_{i:}$ without reducing over $P_{i:}$ and $dP_{i:}$ of size N (they might not fit into SRAM). Instead we can rewrite $D_i = do_i^\top o_i$ 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_{drop}, pseudo-random number generator state $\mathcal{R}$ from the forward pass.

Set the pseudo-random number generator state to $\mathcal{R}$.
Set block sizes $B_c = \left\lceil \frac{M}{4d} \right\rceil$, $B_r = \min\left(\left\lceil \frac{M}{4d} \right\rceil, d\right)$.
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.
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 ℓ 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.
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.
for $1 \le j \le T_c$ **do**
  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
  Initialize $\mathbf{d\tilde{K}}_j = (0)_{B_c \times d}, \mathbf{d\tilde{V}}_j = (0)_{B_c \times d}$ on SRAM.
  for $1 \le i \le T_r$ **do**
    Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
    On chip, compute $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
    On chip, compute $\mathbf{S}_{ij}^{\text{masked}} = \text{MASK}(\mathbf{S}_{ij})$.
    On chip, compute $\mathbf{P}_{ij} = \text{diag}(l_i)^{-1} \exp(\mathbf{S}_{ij}^{\text{masked}} - m_i) \in \mathbb{R}^{B_r \times B_c}$.
    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_{drop}.
    On chip, compute $\mathbf{P}_{ij}^{\text{dropped}} = \mathbf{P}_{ij} \circ \mathbf{Z}_{ij}$ (pointwise multiply).
    On chip, compute $\mathbf{d\tilde{V}}_j \leftarrow \mathbf{d\tilde{V}}_j + (\mathbf{P}_{ij}^{\text{dropped}})^\top \mathbf{dO}_i \in \mathbb{R}^{B_c \times d}$.
    On chip, compute $\mathbf{dP}_{ij}^{\text{dropped}} = \mathbf{dO}_i \mathbf{V}_j^\top \in \mathbb{R}^{B_r \times B_c}$.
    On chip, compute $\mathbf{dP}_{ij} = \mathbf{dP}_{ij}^{\text{dropped}} \circ \mathbf{Z}_{ij}$ (pointwise multiply).
    On chip, compute $D_i = \text{rowsum}(\mathbf{dO}_i \circ \mathbf{O}_i) \in \mathbb{R}^{B_r}$.
    On chip, compute $\mathbf{dS}_{ij} = \mathbf{P}_{ij} \circ (\mathbf{dP}_{ij} - D_i) \in \mathbb{R}^{B_r \times B_c}$.
    Write $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \tau \mathbf{dS}_{ij} \mathbf{K}_j \in \mathbb{R}^{B_r \times d}$ to HBM.
    On chip, compute $\mathbf{d\tilde{K}}_j \leftarrow \mathbf{d\tilde{K}}_j + \tau \mathbf{dS}_{ij}^\top \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$.
  end for
  Write $\mathbf{dK}_j \leftarrow \mathbf{d\tilde{K}}_j, \mathbf{dV}_j \leftarrow \mathbf{d\tilde{V}}_j$ to HBM.
end for
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 $d \le M \le Nd$. Standard attention (Algorithm 0) backward pass requires $\Theta(Nd + N^2)$ HBM accesses, while* FlashAttention *backward pass (Algorithm 4) requires $\Theta(N^2 d^2 M^{-1})$ 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 (2-4×) 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 (ℓ, m).

We now prove the algorithm's correctness by induction on j for $0 \le j \le T_c$. Let $\mathbf{K}_{:j} \in \mathbb{R}^{jB_c \times d}$ be the first jB_c rows of $\mathbf{K}$, and similarly $\mathbf{V}_{:j} \in \mathbb{R}^{jB_c \times d}$ the the first jB_c rows of $\mathbf{V}$. Let $\mathbf{S}_{:,:j} = \mathbf{Q}\mathbf{K}_{:j}^\top \in \mathbb{R}^{N \times jB_c}$, and $\mathbf{P}_{:,:j} = \mathrm{softmax}(\mathbf{S}_{:,:j}) \in \mathbb{R}^{N \times jB_c}$ (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:

$$m^{(j)} = \mathrm{rowmax}(\mathbf{S}_{:,:j}) \in \mathbb{R}^N, \quad \ell^{(j)} = \mathrm{rowsum}(\exp(\mathbf{S}_{:,:j} - m^{(j)})) \in \mathbb{R}^N, \quad \mathbf{O}^{(j)} = \mathbf{P}_{:,:j} \mathbf{V}_{:j} \in \mathbb{R}^{N \times d}.$$

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 $j = 0, \ldots, T_c - 1$. 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

$$m^{(j+1)} = \text{rowmax}(\mathbf{S}_{:,:j+1}) \in \mathbb{R}^N.$$

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:

$$\ell^{(j+1)} = \text{rowsum}(\exp(\mathbf{S}_{:,:j+1} - m^{(j+1)})) \in \mathbb{R}^N.$$

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:

$$\begin{aligned}
\mathbf{O}^{(j+1)} &= \text{diag}(\ell^{(j+1)})^{-1}(\text{diag}(\ell^{(j)})e^{m^{(j)} - m^{(j+1)}}\mathbf{O}^{(j)} + e^{\tilde{m} - m^{(j+1)}} \exp(\mathbf{S}_{j:j+1} - \tilde{m})\mathbf{V}_{j:j+1}) \\
&= \text{diag}(\ell^{(j+1)})^{-1}(\text{diag}(\ell^{(j)})e^{m^{(j)} - m^{(j+1)}}\mathbf{P}_{:,:j}\mathbf{V}_{:j} + e^{-m^{(j+1)}} \exp(\mathbf{S}_{j:j+1})\mathbf{V}_{j:j+1}) \\
&= \text{diag}(\ell^{(j+1)})^{-1}(\text{diag}(\ell^{(j)})e^{m^{(j)} - m^{(j+1)}}\text{diag}(\ell^{(j)}) \exp(\mathbf{S}_{:,:j} - m^{(j)})\mathbf{V}_{:j} + e^{-m^{(j+1)}} \exp(\mathbf{S}_{j:j+1})\mathbf{V}_{j:j+1}) \\
&= \text{diag}(\ell^{(j+1)})^{-1}(e^{-m^{(j+1)}} \exp(\mathbf{S}_{:,:j})\mathbf{V}_{:j} + e^{-m^{(j+1)}} \exp(\mathbf{S}_{j:j+1})\mathbf{V}_{j:j+1}) \\
&= \text{diag}(\ell^{(j+1)})^{-1}(\exp(\mathbf{S}_{:,:j} - m^{(j+1)})\mathbf{V}_{:j} + \exp(\mathbf{S}_{j:j+1} - m^{(j+1)})\mathbf{V}_{j:j+1}) \\
&= \text{diag}(\ell^{(j+1)})^{-1}\left(\exp\left(\begin{bmatrix}\mathbf{S}_{:,:j} & \mathbf{S}_{j:j+1}\end{bmatrix} - m^{(j+1)}\right)\right)\begin{bmatrix}\mathbf{V}_{:j} \\ \mathbf{V}_{j:j+1}\end{bmatrix} \\
&= \text{softmax}(\mathbf{S}_{:j+1})\mathbf{V}_{:j+1}.
\end{aligned}$$

We then see that the claim is also true for $j+1$. By induction, the claim is true for all $j = 0, \ldots, T_c$.

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 the 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 $\Theta\left(Nd + NdT_c\right) = \Theta(NdT_c)$.

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:

$$B_c = \Theta\left(\frac{M}{d}\right), \qquad B_r = \Theta\left(\min\left(\frac{M}{d}, \frac{M}{B_c}\right)\right) = \Theta\left(\min\left(\frac{M}{d}, d\right)\right).$$

We then have:

$$T_c = \frac{N}{B_c} = \Theta\left(\frac{Nd}{M}\right).$$

As a result, the number of HBM accesses is:

$$\Theta\left(NdT_c\right) = \Theta\left(\frac{N^2d^2}{M}\right).$$

□

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 $\Theta\left(Nd + NdT_c\right) = \Theta(NdT_c)$.

As in the proof of Theorem 2, the constraints on the block sizes are that:

$$B_c = \Theta\left(\frac{M}{d}\right), \qquad B_r = \Theta\left(\min\left(\frac{M}{d}, d\right)\right).$$

We then have:

$$T_c = \frac{N}{B_c} = \Theta\left(\frac{Nd}{M}\right).$$

As a result, the number of HBM accesses is:

$$\Theta\left(NdT_c\right) = \Theta\left(\frac{N^2d^2}{M}\right).$$

□

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_{drop}, block sizes $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$, block sparsity mask $M \in \{0,1\}^{N/B_r \times N/B_c}$..

Initialize the pseudo-random number generator state $\mathcal{R}$ and save to HBM.
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.
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.
Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ 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.
for $1 \le j \le T_c$ **do**
  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
  for $1 \le i \le T_r$ **do**
    if $M_{ij} \neq 0$ **then**
      Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
      On chip, compute $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
      On chip, compute $\mathbf{S}_{ij}^{\text{masked}} = \text{MASK}(\mathbf{S}_{ij})$.
      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), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
      On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
      On chip, compute $\tilde{\mathbf{P}}_{ij}^{\text{dropped}} = \text{dropout}(\tilde{\mathbf{P}}_{ij}, p_{\text{drop}})$.
      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.
      Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
    end if
  end for
end for
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

$$\Theta\left(Nd + \frac{N^2 d^2}{M} s\right).$$

□

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. As 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 8×A100-80GB 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 8×A100-40GB 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].

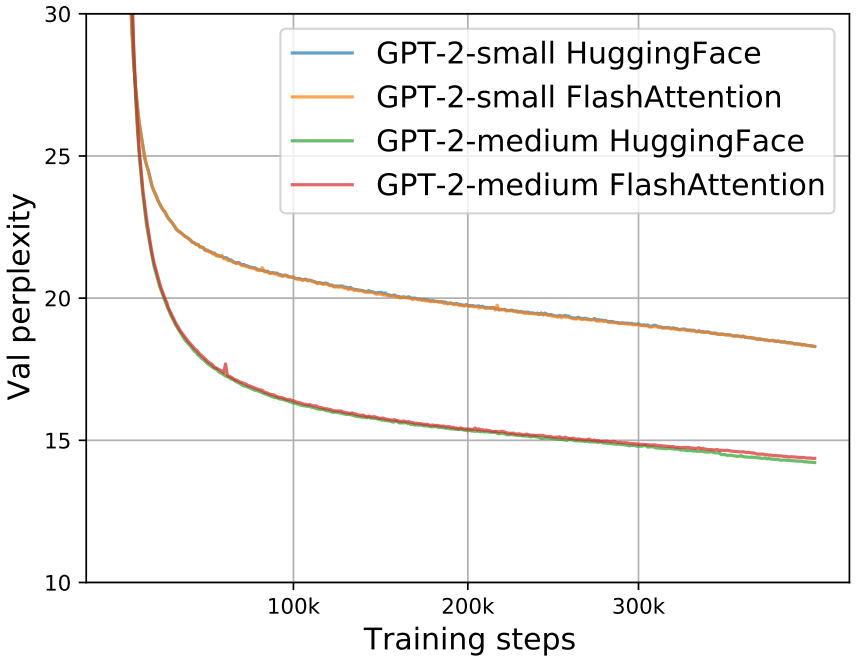


Figure 4: Validation perplexity of GPT-2 small/medium using two implementations. We confirm that FlashAttention yields the same validation curves as the baseline implementation from HuggingFace.

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	**0.08**	**0.22**	**0.81**
Apex FMHA backward	**0.17**	**0.52**	**1.81**
FlashAttention backward	0.20	0.53	2.00
Apex FMHA forward + backward	**0.27**	0.81	2.95
FlashAttention forward + backward	0.28	**0.75**	**2.81**

FMHA targets BERT models, it only supports head dimension 64, and only runs on A100 GPUs. FMHA fuses the attention computation dropout(softmax(mask($\mathbf{QK}^\top$)))$\mathbf{V}$ into one CUDA kernel. In the forward pass, it stores the attention matrix softmax(mask($\mathbf{QK}^T$)) 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.

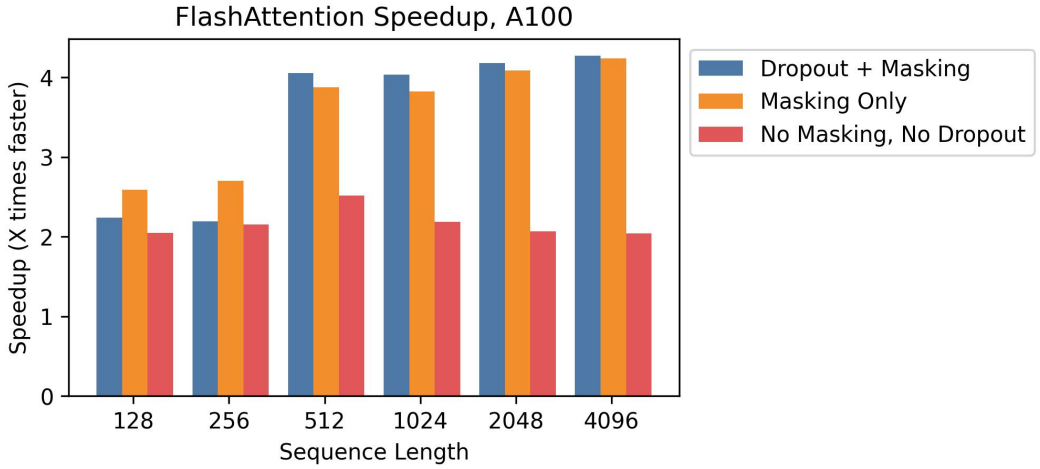


Figure 5: Speedup over standard PyTorch attention at different sequence lengths, on A100.

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.

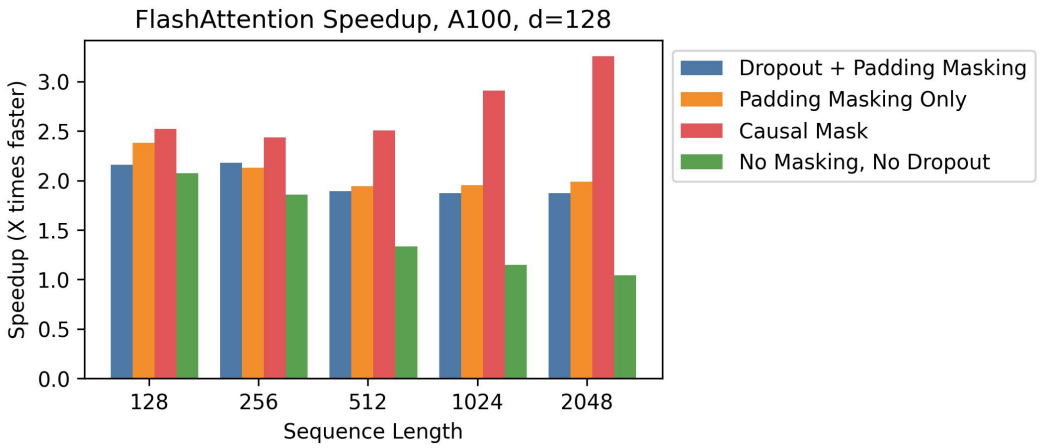


Figure 6: Speedup over standard PyTorch attention at different sequence lengths, on A100, with head dimension 128.

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.

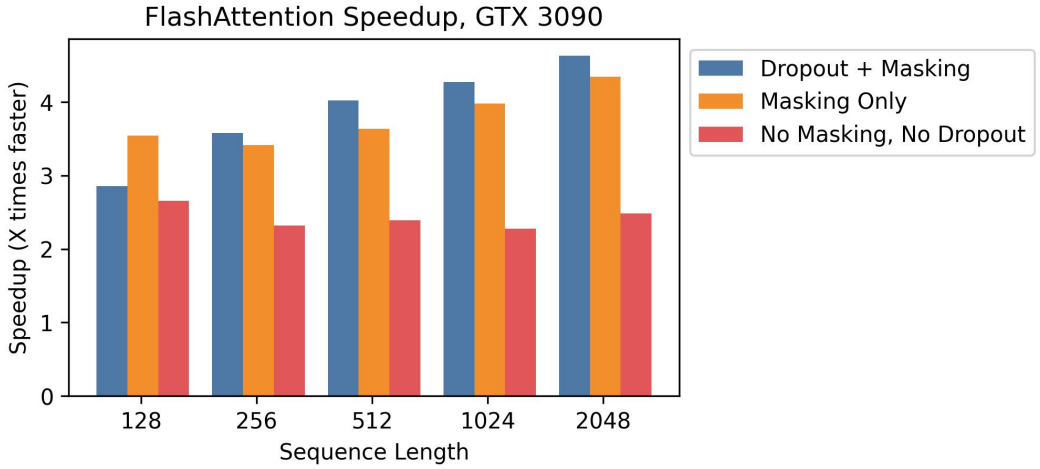


Figure 7: Speedup over standard PyTorch attention at different sequence lengths, on RTX 3090.

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.

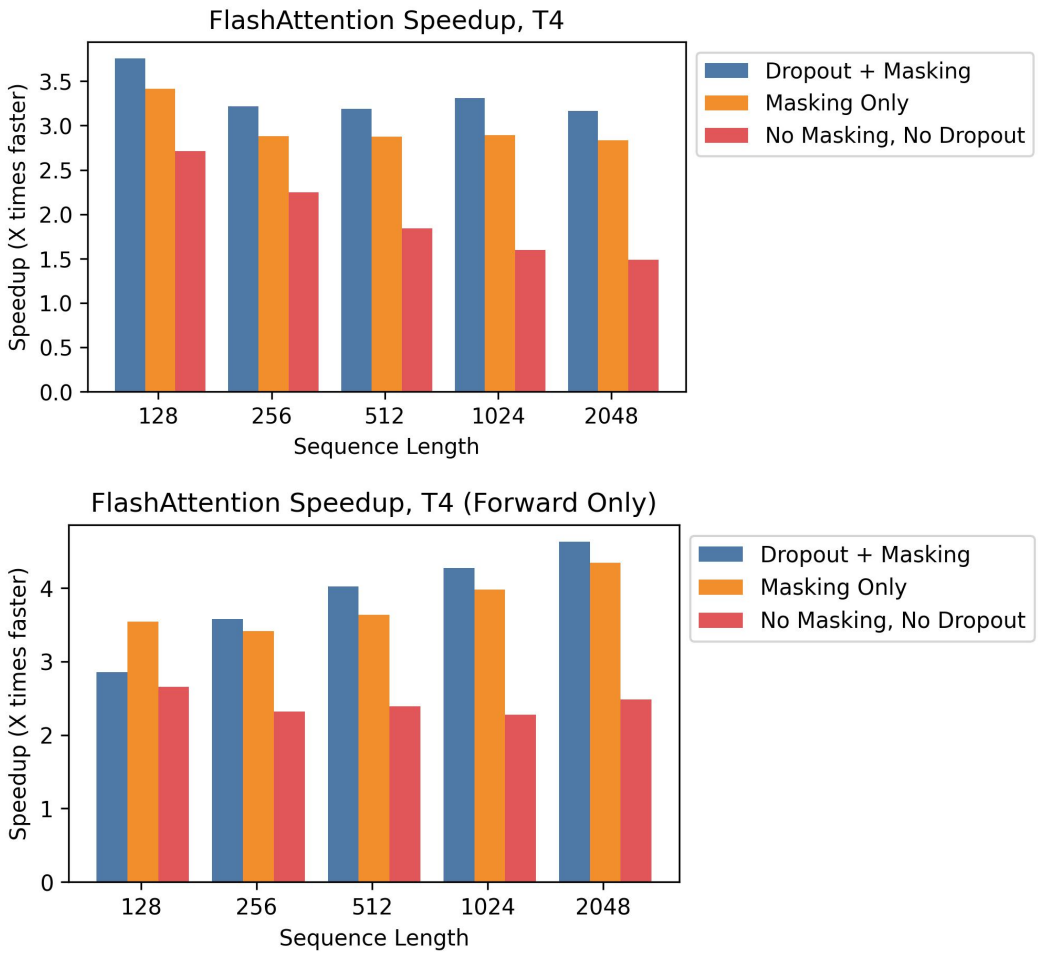


Figure 8: Speedup over standard PyTorch attention at different sequence lengths, on T4. **Top:** Combined forward pass + backward pass. **Bottom:** Forward pass only.

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 form 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
PyTorch Attention	0.36	0.34	0.78	2.54	9.33	36.33	-	-	-	-
Megatron	0.40	0.40	1.10	3.65	16.19	-	-	-	-	-
Reformer	2.03	3.15	5.67	11.02	22.59	46.14	97.38	212.13	-	-
Local Attention	0.83	0.86	1.01	2.20	7.13	14.32	28.60	57.79	117.67	-
Linformer	0.67	0.52	0.69	0.71	1.65	3.18	6.15	12.16	24.17	52.39
Smyrf	2.27	2.34	3.91	7.44	14.71	29.22	58.27	116.41	-	-
LSformer	1.18	1.27	1.34	3.38	11.40	22.55	44.95	89.76	179.66	-
Block Sparse	1.12	1.11	2.13	2.77	6.95	20.91	-	-	-	-
Longformer	1.22	1.14	1.08	1.95	5.72	12.98	-	-	-	-
BigBird	1.13	1.12	1.12	1.77	6.03	13.68	-	-	-	-
FlashAttention	**0.04**	0.06	0.21	0.82	2.85	10.41	41.74	167.19	670.76	2682.35
Block-Sparse FlashAttention	0.06	**0.06**	**0.06**	**0.12**	**0.44**	**0.86**	**1.70**	**3.29**	**6.55**	**13.34**

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	87.67	-	-	-	-
Megatron	0.35	0.32	0.77	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	0.93	2.48	4.75	9.29	18.27	36.53	-
Smyrf	1.41	2.83	5.43	10.72	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	-	-	-
FlashAttention	**0.30**	**0.26**	0.68	2.02	6.84	26.89	105.70	418.96	1666.89	6660.44
Block-Sparse FlashAttention	**0.30**	0.27	**0.29**	**0.59**	**1.50**	**2.94**	**5.82**	**11.85**	**23.98**	**47.61**

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	1.60	4.19	8.04	15.71	30.92	61.47	-
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	-	-	-
FlashAttention	**0.43**	**0.41**	0.95	2.55	9.56	37.49	147.75	586.61	2339.11	9341.30
Block-Sparse FlashAttention	0.44	0.44	**0.45**	**0.89**	**1.95**	**4.12**	**7.64**	**16.60**	**32.73**	**64.11**

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	0.65	1.36	2.60	5.04	9.92	19.69	43.47
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	-	-	-
FlashAttention	**0.03**	**0.04**	0.17	0.68	2.28	8.40	33.55	134.14	537.50	2150.88
Block-Sparse FlashAttention	0.05	**0.04**	**0.05**	**0.11**	**0.35**	**0.68**	**1.33**	**2.54**	**5.34**	**10.73**

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	0.87	2.24	4.25	8.35	16.38	32.67	72.11
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	-	-	-
FlashAttention	**0.21**	**0.22**	0.62	1.84	5.77	22.25	86.21	338.91	1343.91	5361.09
Block-Sparse FlashAttention	0.22	0.22	**0.26**	**0.57**	**1.55**	**3.13**	**5.98**	**12.21**	**23.49**	**47.85**

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	-	-	-	-
Megatron	0.81	0.83	1.09	3.36	12.39	-	-	-	-	-
Reformer	4.16	7.46	14.06	27.68	55.66	112.15	229.37	-	-	-
Local Attention	1.39	1.68	2.08	5.83	20.04	40.16	80.44	161.35	325.11	-
Linformer	1.51	1.42	1.56	1.67	3.67	6.99	13.63	26.77	53.36	117.56
Smyrf	3.38	4.93	9.07	17.66	34.94	69.55	138.72	277.41	-	-
LSformer	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	-	-	-	-
Longformer	2.36	2.34	2.38	2.94	9.83	21.35	58.12	-	-	-
BigBird	2.35	2.35	2.37	3.25	10.36	22.57	60.63	-	-	-
FlashAttention	**0.32**	**0.30**	0.83	2.37	7.95	30.77	119.98	473.65	1883.43	7513.01
Block-Sparse FlashAttention	0.34	0.34	**0.36**	**0.69**	**1.85**	**3.89**	**7.16**	**14.85**	**30.46**	**60.03**

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	-	-	-	-
Megatron	0.27	0.27	0.56	1.88	6.56	-	-	-	-	-
Reformer	1.83	2.96	5.31	10.33	21.19	43.42	91.96	201.34	-	-
Local Attention	0.51	0.60	0.78	2.01	6.23	12.52	25.07	50.50	102.18	-
Linformer	0.47	0.37	0.49	**0.52**	1.37	2.65	5.12	10.13	20.25	44.16
Smyrf	2.12	2.01	3.15	5.97	11.83	23.36	46.48	92.72	-	-
LSformer	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	-	-	-	-
Longformer	1.02	1.03	1.03	1.73	5.10	11.63	34.22	-	-	-
BigBird	0.99	1.03	1.01	1.58	5.36	12.27	35.56	-	-	-
FlashAttention	**0.10**	**0.10**	**0.22**	0.83	2.81	10.38	41.63	167.01	668.74	2678.11
Block-Sparse FlashAttention	0.54	0.51	0.68	0.61	**0.67**	**1.10**	**1.89**	**3.71**	**7.18**	**14.41**

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	-	-	-	-
Megatron	0.28	0.33	0.92	2.94	10.80	-	-	-	-	-
Reformer	2.24	4.34	8.39	16.62	33.02	65.77	131.52	-	-	-
Local Attention	0.51	0.58	1.41	3.71	12.96	25.98	51.94	103.72	207.78	-
Linformer	0.84	0.74	0.79	0.85	2.28	4.37	8.66	17.02	33.78	-
Smyrf	1.27	2.56	4.90	9.66	19.16	38.13	76.17	152.39	-	-
LSformer	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	-	-	-	-
Longformer	1.28	1.34	1.38	1.98	5.24	10.74	25.95	-	-	-
BigBird	1.48	1.47	1.50	1.81	5.57	11.38	27.43	-	-	-
FlashAttention	**0.15**	0.18	0.58	1.86	6.50	26.21	104.27	416.10	1661.92	6643.01
Block-Sparse FlashAttention	0.17	**0.17**	**0.17**	**0.40**	**1.10**	**2.04**	**4.43**	**9.33**	**18.28**	**37.31**

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	0.67	1.43	4.82	17.47	67.29	-	-	-	-
Megatron	0.88	0.90	1.49	4.73	17.41	-	-	-	-	-
Reformer	4.06	7.28	13.68	26.98	54.27	109.39	223.80	-	-	-
Local Attention	1.09	1.40	1.99	5.61	19.23	38.62	77.30	154.63	311.12	-
Linformer	1.31	1.21	1.30	1.39	3.73	7.15	14.05	27.69	55.00	-
Smyrf	3.00	4.37	8.05	15.66	31.04	61.64	123.04	245.65	-	-
LSformer	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	-	-	-	-
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	-	-	-
FlashAttention	**0.35**	**0.36**	**0.80**	2.52	9.16	36.70	146.13	583.45	2332.01	9323.63
Block-Sparse FlashAttention	0.91	0.83	0.94	**0.92**	**1.83**	**3.50**	**7.02**	**13.56**	**26.71**	**53.92**

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	0.21	0.22	0.43	1.27	4.32	16.47	67.77	-	-	-
Megatron	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	-	-
Local Attention	0.48	0.57	0.80	1.90	5.76	11.56	23.13	46.65	94.74	-
Linformer	0.46	0.36	0.45	**0.50**	1.09	2.09	4.01	7.90	15.70	35.40
Smyrf	1.94	1.96	3.01	5.69	11.26	22.23	44.21	88.22	-	-
LSformer	1.21	1.34	1.34	3.31	11.01	21.71	43.27	86.32	172.85	-
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	-	-	-
FlashAttention	**0.08**	**0.09**	**0.18**	0.68	2.40	8.42	33.54	134.03	535.95	2147.05
Block-Sparse FlashAttention	0.56	0.52	0.63	0.65	**0.61**	**0.96**	**1.69**	**3.02**	**5.69**	**11.77**

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	-	-	-	-
Megatron	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	-	-	-
Local Attention	0.51	0.64	1.28	3.60	12.52	25.08	50.22	100.23	200.66	-
Linformer	0.69	0.76	0.69	0.80	2.04	3.88	7.67	15.04	30.11	63.15
Smyrf	1.24	2.49	4.77	9.42	18.65	37.12	74.15	148.35	-	-
LSformer	1.68	1.61	3.02	7.40	19.72	38.27	74.89	147.99	-	-
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	-	-	-
FlashAttention	**0.11**	0.16	0.52	1.62	5.45	21.57	84.75	336.00	1338.56	5343.19
Block-Sparse FlashAttention	0.11	**0.12**	**0.16**	**0.38**	**1.20**	**2.34**	**4.69**	**9.10**	**18.74**	**37.04**

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	-	-	-	-
Megatron	0.74	0.65	1.23	3.80	13.21	-	-	-	-	-
Reformer	3.93	7.01	13.15	25.89	52.09	105.00	215.13	-	-	-
Local Attention	1.09	1.27	1.99	5.38	18.32	36.77	73.67	147.29	296.35	-
Linformer	1.31	1.25	1.30	1.29	3.20	6.10	11.93	23.39	46.72	100.52
Smyrf	2.98	4.23	7.78	15.12	29.96	59.45	118.60	237.02	-	-
LSformer	3.03	3.05	4.26	10.70	30.77	60.15	118.33	234.94	-	-
Block Sparse	2.39	2.40	3.31	5.02	12.25	35.94	-	-	-	-
Longformer	2.36	2.34	2.38	2.94	9.83	21.35	58.12	-	-	-
BigBird	2.35	2.35	2.37	3.25	10.36	22.57	60.63	-	-	-
FlashAttention	**0.31**	**0.31**	**0.73**	2.29	7.64	30.09	118.50	470.51	1876.08	7492.85
Block-Sparse FlashAttention	0.74	0.77	0.82	**0.88**	**1.71**	**3.21**	**6.56**	**12.60**	**24.93**	**50.39**

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	-	-	-	-
Megatron	36	104	336	1184	4416	-	-	-	-	-
Reformer	377	754	1508	3016	6033	12067	24134	-	-	-
Local Attention	53	110	232	592	1696	3392	6784	13568	27136	-
Linformer	25	52	114	287	832	1652	3292	6572	13132	26252
Smyrf	217	434	868	1737	3474	6947	13894	27788	-	-
LSformer	72	152	333	796	2540	5068	10125	20240	-	-
Block Sparse	33	82	228	408	910	2401	-	-	-	-
Longformer	30	61	124	277	681	1370	2748	-	-	-
BigBird	33	66	131	294	708	1431	2872	-	-	-
FlashAttention	**22**	**44**	**104**	**209**	**418**	**836**	**1672**	**3344**	**6688**	**13376**
Block-Sparse FlashAttention	22	44	104	209	418	836	1672	3344	6690	13384

FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning

Tri Dao[1,2]

[1]Department of Computer Science, Princeton University
[2]Department of Computer Science, Stanford University
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).[1]

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,

[1]FlashAttention-2 is available at https://github.com/Dao-AILab/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× 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× 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× 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 $\mathbf{O} \in \mathbb{R}^{N \times d}$:

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

where softmax is applied row-wise.[2] 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):

$$\begin{aligned}
\mathbf{dV} &= \mathbf{P}^\top \mathbf{dO} \in \mathbb{R}^{N \times d} \\
\mathbf{dP} &= \mathbf{dO}\mathbf{V}^\top \in \mathbb{R}^{N \times N} \\
\mathbf{dS} &= \mathrm{dsoftmax}(\mathbf{dP}) \in \mathbb{R}^{N \times N} \\
\mathbf{dQ} &= \mathbf{dS}\mathbf{K} \in \mathbb{R}^{N \times d} \\
\mathbf{dK} &= \mathbf{Q}\mathbf{dS}^\top \in \mathbb{R}^{N \times d},
\end{aligned}$$

where dsoftmax is the gradient (backward pass) of softmax applied row-wise. One can work out that if $p = \mathrm{softmax}(s)$ for some vector s and p, then with output gradient dp, the input gradient $ds = (\mathrm{diag}(p) - pp^\top)dp$.

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 § 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 $\begin{bmatrix}\mathbf{S}^{(1)} & \mathbf{S}^{(2)}\end{bmatrix}$ 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

[2] 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:

$$
\begin{aligned}
m &= \mathrm{max}(\mathrm{rowmax}(\mathbf{S}^{(1)}), \mathrm{rowmax}(\mathbf{S}^{(2)})) \in \mathbb{R}^{B_r} \\
\ell &= \mathrm{rowsum}(e^{\mathbf{S}^{(1)}-m}) + \mathrm{rowsum}(e^{\mathbf{S}^{(2)}-m}) \in \mathbb{R}^{B_r} \\
\mathbf{P} &= \begin{bmatrix}\mathbf{P}^{(1)} & \mathbf{P}^{(2)}\end{bmatrix} = \mathrm{diag}(\ell)^{-1}\begin{bmatrix}e^{\mathbf{S}^{(1)}-m} & e^{\mathbf{S}^{(2)}-m}\end{bmatrix} \in \mathbb{R}^{B_r \times 2B_c} \\
\mathbf{O} &= \begin{bmatrix}\mathbf{P}^{(1)} & \mathbf{P}^{(2)}\end{bmatrix}\begin{bmatrix}\mathbf{V}^{(1)} \\ \mathbf{V}^{(2)}\end{bmatrix} = \mathrm{diag}(\ell)^{-1}e^{\mathbf{S}^{(1)}-m}\mathbf{V}^{(1)} + e^{\mathbf{S}^{(2)}-m}\mathbf{V}^{(2)} \in \mathbb{R}^{B_r \times d}.
\end{aligned}
$$

Online softmax instead computes "local" softmax with respect to each block and rescale to get the right output at the end:

$$
\begin{aligned}
m^{(1)} &= \mathrm{rowmax}(\mathbf{S}^{(1)}) \in \mathbb{R}^{B_r} \\
\ell^{(1)} &= \mathrm{rowsum}(e^{\mathbf{S}^{(1)}-m^{(1)}}) \in \mathbb{R}^{B_r} \\
\tilde{\mathbf{P}}^{(1)} &= \mathrm{diag}(\ell^{(1)})^{-1}e^{\mathbf{S}^{(1)}-m^{(1)}} \in \mathbb{R}^{B_r \times B_c} \\
\mathbf{O}^{(1)} &= \tilde{\mathbf{P}}^{(1)}\mathbf{V}^{(1)} = \mathrm{diag}(\ell^{(1)})^{-1}e^{\mathbf{S}^{(1)}-m^{(1)}}\mathbf{V}^{(1)} \in \mathbb{R}^{B_r \times d} \\
m^{(2)} &= \mathrm{max}(m^{(1)}, \mathrm{rowmax}(\mathbf{S}^{(2)})) = m \\
\ell^{(2)} &= e^{m^{(1)}-m^{(2)}}\ell^{(1)} + \mathrm{rowsum}(e^{\mathbf{S}^{(2)}-m^{(2)}}) = \mathrm{rowsum}(e^{\mathbf{S}^{(1)}-m}) + \mathrm{rowsum}(e^{\mathbf{S}^{(2)}-m}) = \ell \\
\tilde{\mathbf{P}}^{(2)} &= \mathrm{diag}(\ell^{(2)})^{-1}e^{\mathbf{S}^{(2)}-m^{(2)}} \\
\mathbf{O}^{(2)} &= \mathrm{diag}(\ell^{(1)}/\ell^{(2)})^{-1}\mathbf{O}^{(1)} + \tilde{\mathbf{P}}^{(2)}\mathbf{V}^{(2)} = \mathrm{diag}(\ell^{(2)})^{-1}e^{s^{(1)}-m}\mathbf{V}^{(1)} + \mathrm{diag}(\ell^{(2)})^{-1}e^{s^{(2)}-m}\mathbf{V}^{(2)} = \mathbf{O}.
\end{aligned}
$$

We show how FLASHATTENTION uses online softmax to enable tiling (Fig. 1) to reduce memory reads/writes.

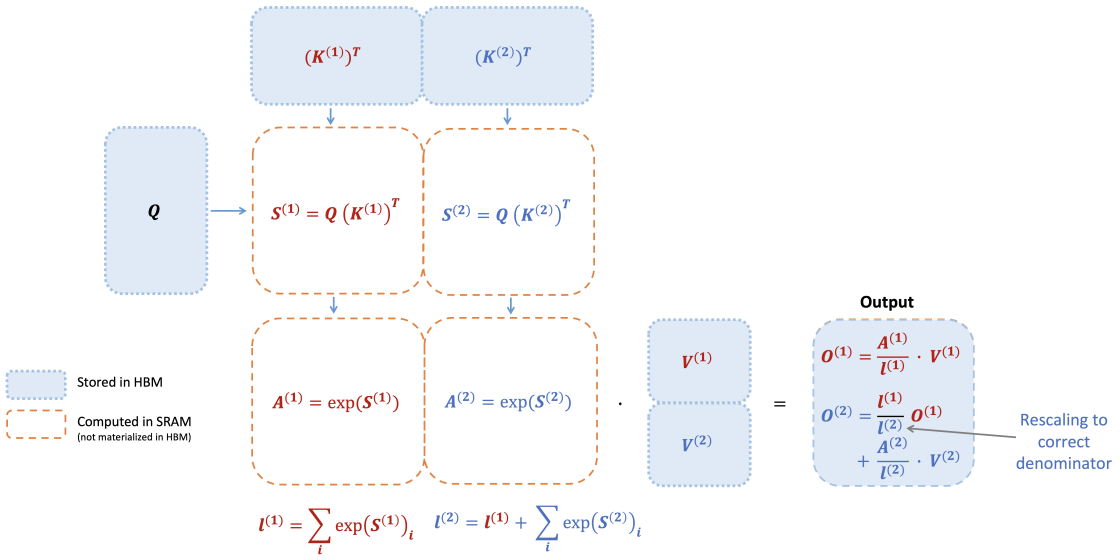


Figure 1: Diagram of how FLASHATTENTION forward pass is performed, when the key $\mathbf{K}$ is partitioned into two blocks and the value $\mathbf{V}$ is also partitioned into two blocks. By computing attention with respect to each block and rescaling the output, we get the right answer at the end, while 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× memory saving depending on sequence length (memory required in linear in sequence length N instead of quadratic). The backward pass also achieves 2-4× 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 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× 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× 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 $\mathrm{diag}(\ell^{(2)})^{-1}$:

$$\mathbf{O}^{(2)} = \mathrm{diag}(\ell^{(1)}/\ell^{(2)})^{-1}\mathbf{O}^{(1)} + \mathrm{diag}(\ell^{(2)})^{-1}e^{\mathbf{S}^{(2)}-m^{(2)}}\mathbf{V}^{(2)}.$$

   We can instead maintain an "un-scaled" version of $\mathbf{O}^{(2)}$ and keep around the statistics $\ell^{(2)}$:

$$\tilde{\mathbf{O}}^{(2)} = \mathrm{diag}(\ell^{(1)})^{-1}\mathbf{O}^{(1)} + e^{\mathbf{S}^{(2)}-m^{(2)}}\mathbf{V}^{(2)}.$$

   Only at the every end of the loop do we scale the final $\tilde{\mathbf{O}}^{(\mathrm{last})}$ by $\mathrm{diag}(\ell^{(\mathrm{last})})^{-1}$ to get the right output.

2. We do not have to save both the max $m^{(j)}$ and the sum of exponentials $\ell^{(j)}$ for the backward pass. We only need to store the logsumexp $L^{(j)} = m^{(j)} + \log(\ell^{(j)})$.

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 $\mathbf{K}, \mathbf{V}$ 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 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** $1 \le i \le T_r$ **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** $1 \le j \le T_c$ **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 $O(N^2 d)$ 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.

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.
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.
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.
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.
for $1 \le j \le T_c$ **do**
  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
  Initialize $\mathbf{dK}_j = (0)_{B_c \times d}, \mathbf{dV}_j = (0)_{B_c \times d}$ on SRAM.
  for $1 \le i \le T_r$ **do**
    Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, L_i, D_i$ from HBM to on-chip SRAM.
    On chip, compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
    On chip, compute $\mathbf{P}_i^{(j)} = \exp(\mathbf{S}_{ij} - L_i) \in \mathbb{R}^{B_r \times B_c}$.
    On chip, compute $\mathbf{dV}_j \leftarrow \mathbf{dV}_j + (\mathbf{P}_i^{(j)})^\top \mathbf{dO}_i \in \mathbb{R}^{B_c \times d}$.
    On chip, compute $\mathbf{dP}_i^{(j)} = \mathbf{dO}_i \mathbf{V}_j^\top \in \mathbb{R}^{B_r \times B_c}$.
    On chip, compute $\mathbf{dS}_i^{(j)} = \mathbf{P}_i^{(j)} \circ (\mathbf{dP}_i^{(j)} - D_i) \in \mathbb{R}^{B_r \times B_c}$.
    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.
    On chip, compute $\mathbf{dK}_j \leftarrow \mathbf{dK}_j + \mathbf{dS}_i^{(j)\top} \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$.
  end for
  Write $\mathbf{dK}_j, \mathbf{dV}_j$ to HBM.
end for
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 $\cdot$ 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 ≥ 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.[3]

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.

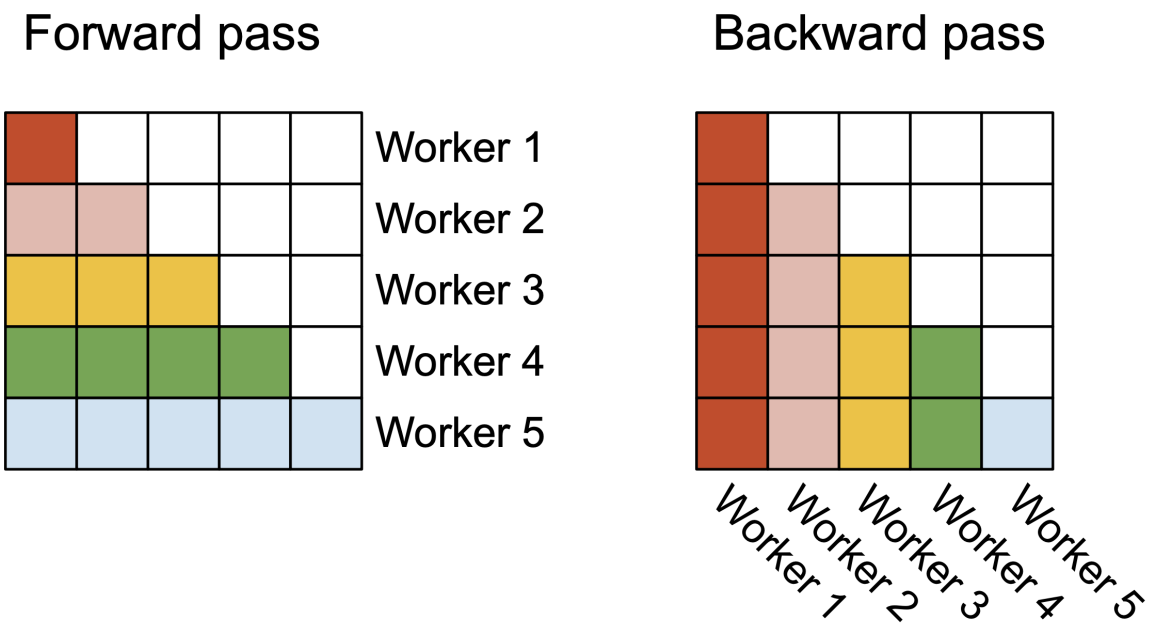


Figure 2: In the forward pass (left), we parallelize the workers (thread blocks) where each worker takes care of a block of rows of the attention matrix. In the backward pass (right), each worker takes care of a block of columns of the attention matrix.

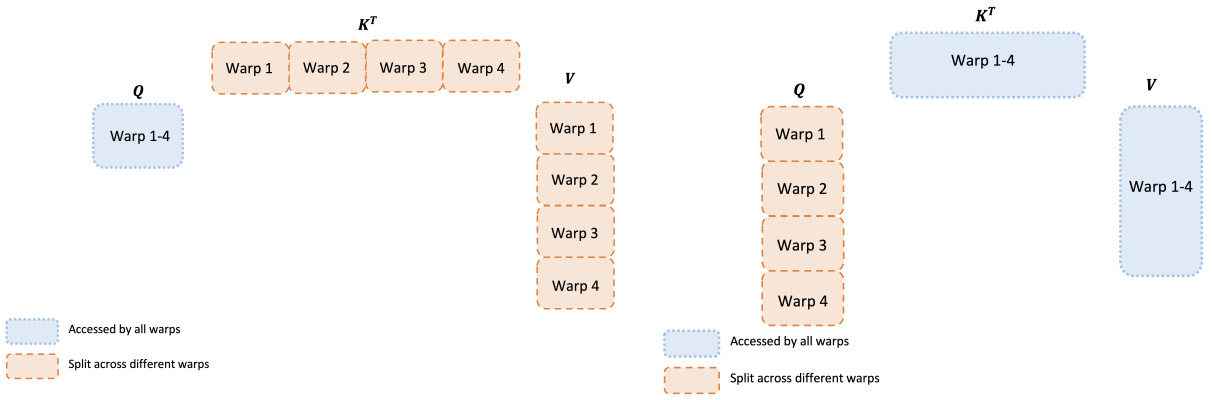
[3] https://github.com/openai/triton/blob/main/python/tutorials/06-fused-attention.py

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 $\mathbf{QK}^\top$, 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 $\mathbf{QK}^\top$, 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).



(a) FlashAttention (b) FlashAttention-2

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× faster than FlashAttention, 1.3-2.5× faster than FlashAttention in Triton, and 3-10× 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× speedup compared to FlashAttention and 2.8× speedup compared to a baseline without FlashAttention. FlashAttention-2 reaches up to 225 TFLOPs/s (72% model FLOPs utilization) per A100 GPU.

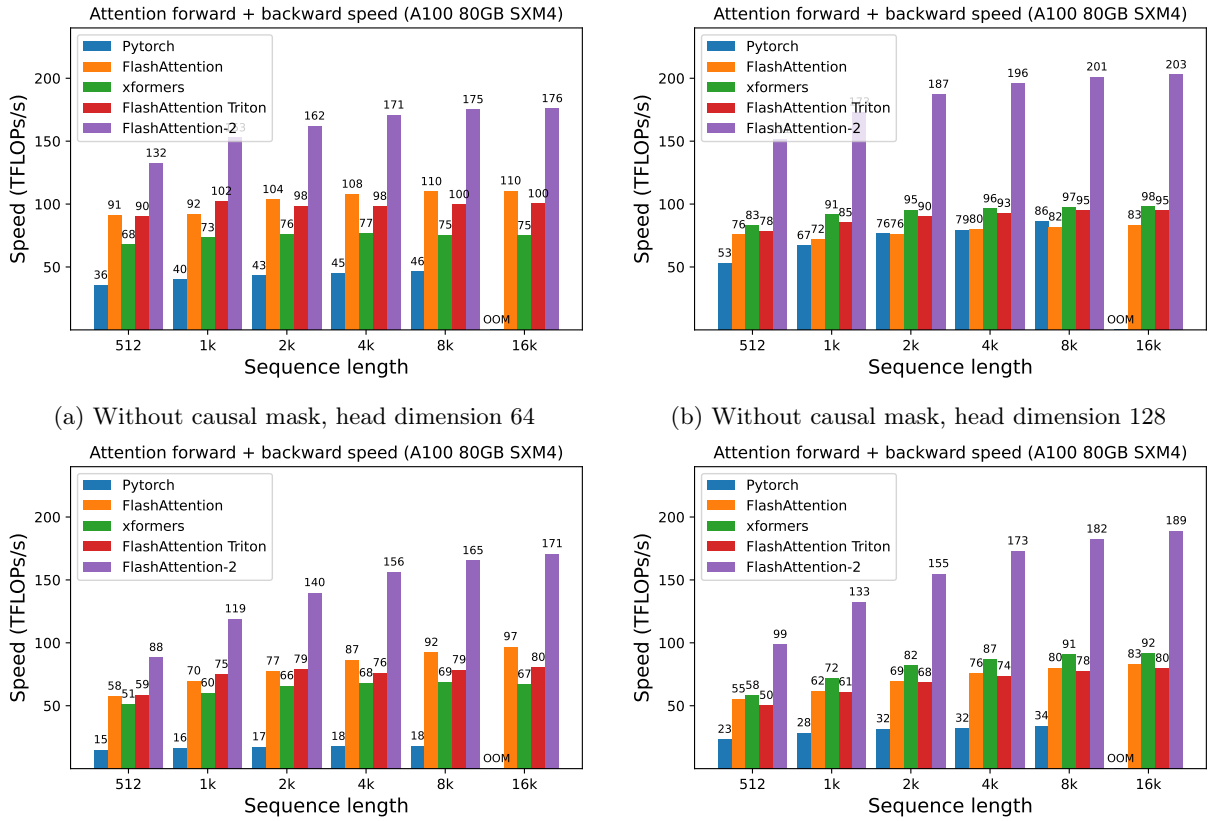
4.1 Benchmarking Attention

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× faster than FlashAttention and FlashAttention in `xformers` (the "cutlass" implementation). FlashAttention-2 is around 1.3-1.5× faster than FlashAttention in Triton in the forward pass and around 2× faster in the backward pass. Compared to a standard attention implementation in PyTorch, FlashAttention-2 can be up to 10× 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).



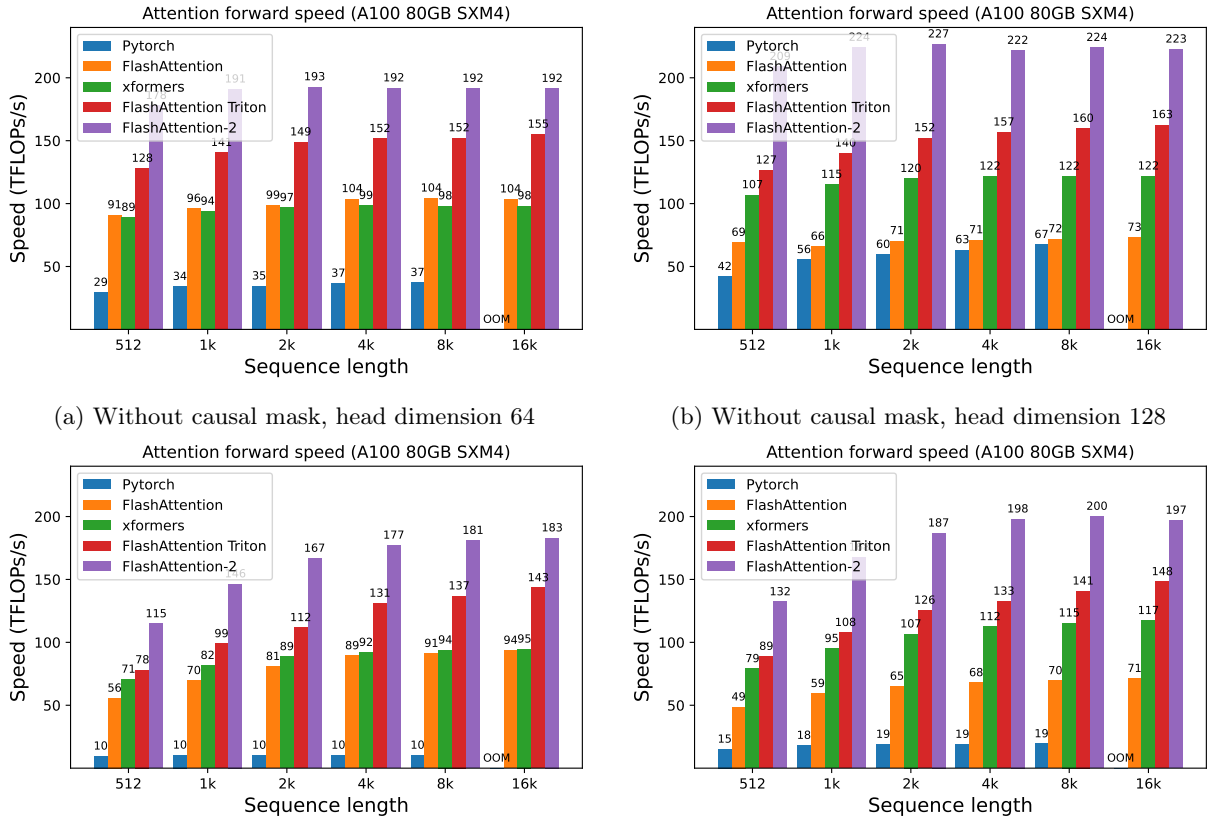
(a) Without causal mask, head dimension 64

(b) Without causal mask, head dimension 128

(c) With causal mask, head dimension 64

(d) With causal mask, head dimension 128

Figure 4: Attention forward + backward speed on A100 GPU



(a) Without causal mask, head dimension 64

(b) Without causal mask, head dimension 128

(c) With causal mask, head dimension 64

(d) With causal mask, head dimension 128

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.

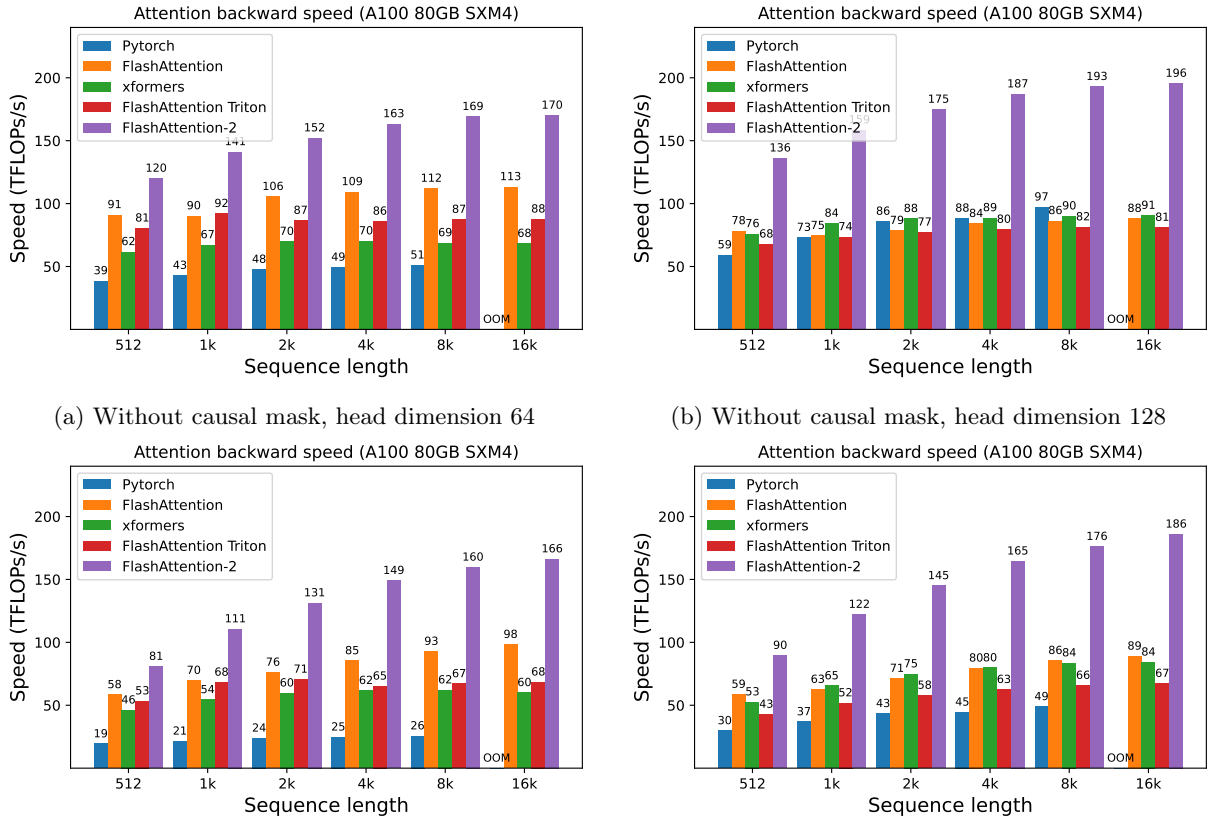
Note that we calculate the FLOPs by the formula, following Megatron-LM [16] (and many other papers and libraries):

$$6 \cdot \text{seqlen} \cdot \text{number of params} + 12 \cdot \text{number of layers} \cdot \text{hidden dim} \cdot \text{seqlen}^2.$$

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.



(a) Without causal mask, head dimension 64

(b) Without causal mask, head dimension 128

(c) With causal mask, head dimension 64

(d) With causal mask, head dimension 128

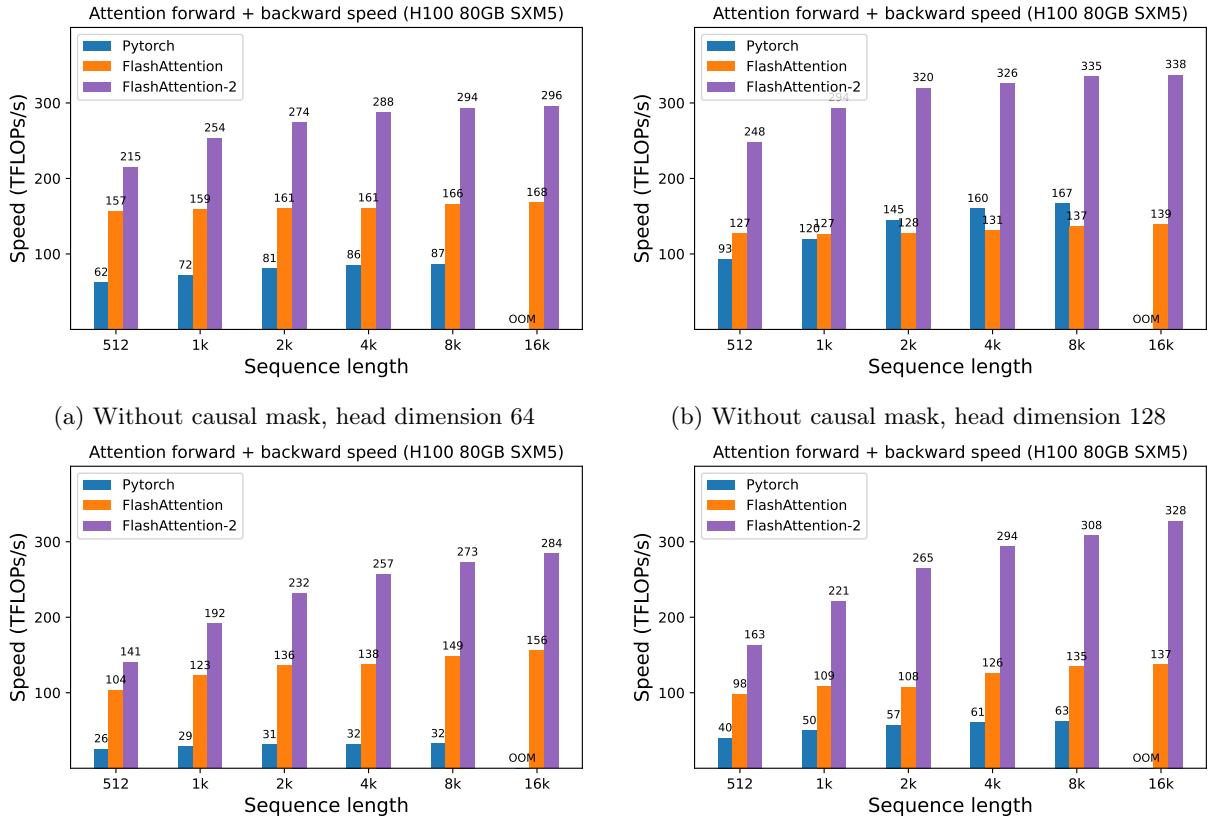
Figure 6: Attention backward speed on A100 GPU

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.



(a) Without causal mask, head dimension 64

(b) Without causal mask, head dimension 128

(c) With causal mask, head dimension 64

(d) With causal mask, head dimension 128

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 Likhosherstov, 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 Lefaudeux, 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, 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.

[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.

FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU

Ying Sheng [1] **Lianmin Zheng** [2] **Binhang Yuan** [3] **Zhuohan Li** [2] **Max Ryabinin** [4 5] **Daniel Y. Fu** [1] **Zhiqiang Xie** [1] **Beidi Chen** [6 7] **Clark Barrett** [1] **Joseph E. Gonzalez** [2] **Percy Liang** [1] **Christopher Ré** [1] **Ion Stoica** [2] **Ce Zhang** [3]

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/FMInference/FlexGen.

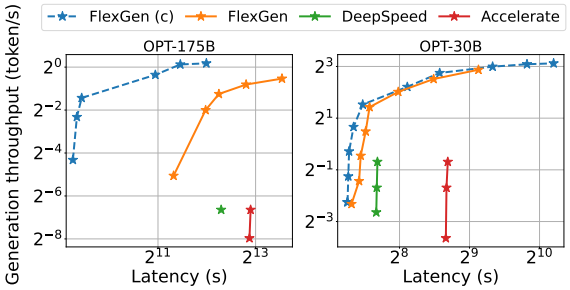


Figure 1. The total latency for a block and throughput trade-offs of three offloading-based systems for OPT-175B (left) and OPT-30B (right) on a single NVIDIA T4 (16 GB) GPU with 208 GB CPU DRAM and 1.5TB SSD. FlexGen achieves a new Pareto-optimal frontier with 100× higher maximum throughput for OPT-175B. 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

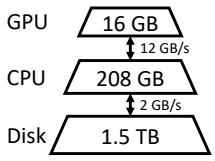
[1]Stanford University [2]UC Berkeley [3]ETH Zurich [4]Yandex [5]HSE University [6]Meta [7]Carnegie Mellon University. Correspondence to: Ying Sheng <ying1123@stanford.edu>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

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 $\mathbf{w}_K^i, \mathbf{w}_Q^i, \mathbf{w}_V^i, \mathbf{w}_O^i \in \mathcal{R}^{h_1 \times h_1}$, $\mathbf{w}_1 \in \mathcal{R}^{h_1 \times h_2}$, and $\mathbf{w}_2 \in \mathcal{R}^{h_2 \times h_1}$.

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:

$$\mathbf{x}_Q^i = \mathbf{x}^i \cdot \mathbf{w}_Q^i$$
$$\mathbf{x}_{\text{Out}}^i = f_{\text{Softmax}}\left(\frac{\mathbf{x}_Q^i {\mathbf{x}_K^i}^T}{\sqrt{h}}\right) \cdot \mathbf{x}_V^i \cdot \mathbf{w}_O^i + \mathbf{x}^i$$
$$\mathbf{x}^{i+1} = f_{\text{relu}}\left(\mathbf{x}_{\text{Out}}^i \cdot \mathbf{w}_1\right) \cdot \mathbf{w}_2 + \mathbf{x}_{\text{Out}}^i$$

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:

$$\mathbf{x}_K^i \leftarrow \mathsf{Concat}\left(\mathbf{x}_K^i, \mathbf{t}^i \cdot \mathbf{w}_K^i\right)$$
$$\mathbf{x}_V^i \leftarrow \mathsf{Concat}\left(\mathbf{x}_V^i, \mathbf{t}^i \cdot \mathbf{w}_V^i\right)$$

and ii) compute the output of the current layer:

$$\mathbf{t}_Q^i = \mathbf{t}^i \cdot \mathbf{w}_Q^i$$
$$\mathbf{t}_{\mathsf{Out}}^i = f_{\mathsf{Softmax}}\left(\frac{\mathbf{t}_Q^i \mathbf{x}_K^{i\,T}}{\sqrt{h}}\right) \cdot \mathbf{x}_V^i \cdot \mathbf{w}_O^i + \mathbf{t}^i$$
$$\mathbf{t}^{i+1} = f_{\mathsf{relu}}\left(\mathbf{t}_{\mathsf{Out}}^i \cdot \mathbf{w}_1\right) \cdot \mathbf{w}_2 + \mathbf{t}_{\mathsf{Out}}^i$$

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 [1] 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.

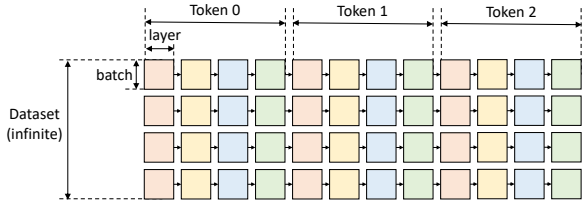


Figure 2. Computational graph of LLM inference.

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.

[1] 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-

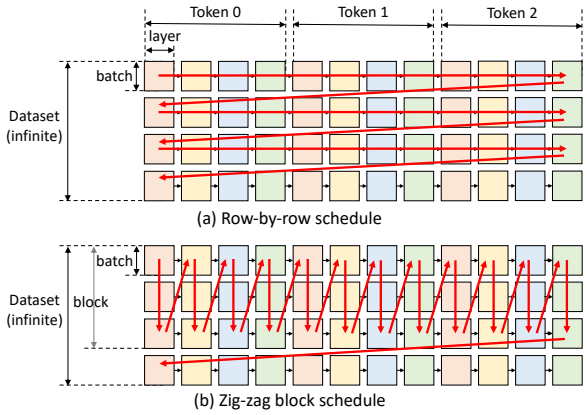


Figure 3. Two different schedules. The red arrows denote the computation order.

Algorithm 1 Block Schedule with Overlapping

```
for i = 1 to generation_length do
  for j = 1 to num_layers do
    // Compute a block with multiple GPU batches
    for k = 1 to num_GPU_batches do
      // Load the weight of the next layer
      load_weight(i, j + 1, k)
      // Store the cache and activation of the prev batch
      store_activation(i, j, k − 1)
      store_cache(i, j, k − 1)
      // Load the cache and activation of the next batch
      load_cache(i, j, k + 1)
      load_activation(i, j, k + 1)
      // Compute this batch
      compute(i, j, k)
      // Synchronize all devices
      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_{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.

$$\begin{aligned} \min_{p} \quad & T/bls \\ \text{s.t.} \quad & gpu\ peak\ memory < gpu\ mem\ capacity \\ & cpu\ peak\ memory < cpu\ mem\ capacity \\ & disk\ peak\ memory < disk\ mem\ capacity \\ & wg + wc + wd = 1 \\ & cg + cc + cd = 1 \\ & hg + hc + hd = 1 \end{aligned} \tag{1}$$

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 $x_{quant} = round\left(\frac{x-min}{max-min} \times (2^b - 1)\right)$.

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 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.

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

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 32×8, 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, 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.

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		
Model size	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		
Model size	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

Runtime breakdown. We shows 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 48×3. "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 × #GPU-batch, wg, wc). More see Appendix A.4.

Model size	30B	175B
All optimizations	7.32 (48×3, 20, 80)	0.69 (32×8, 0, 50)
No policy search	7.26 (48×3, 0, 100)	0.27 (32×1, 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 (ppl)		
Config	FP16	4-bit	4-bit-S	FP16	4-bit	4-bit-S
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[2]. 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.

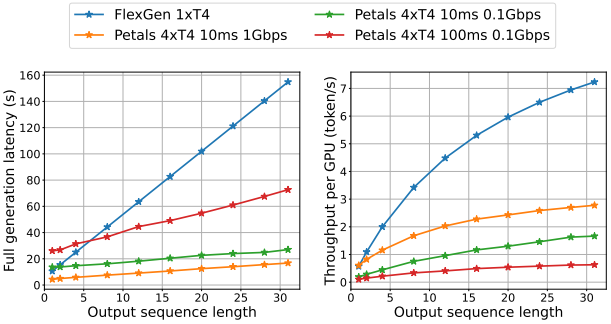


Figure 4. Full latency and per-GPU throughput of FlexGen and Petals in different network delay and bandwidth.

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.

[2]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., Bosselut, 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., Hoefler, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Hoefler, 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. Gpipe: 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. Pagecachemangagement. 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 call bls as the block size as introduced in Section 4.2. The computed $bls \cdot n \cdot l$ squares are called a block.

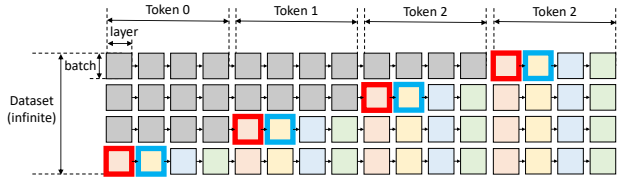


Figure 5. diagonal block schedule

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 peak_mem $<$ CPU memory - some overhead. Let $cmem$ denote the right hand, there is

$$bls \le \frac{cmem - w}{2h_1 + 4h_1 \cdot l \cdot (s+n)} = bls_1$$

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

$$\text{peak\_mem} = w + 2h_1 \cdot bls + \frac{4h_1 \cdot bls \cdot l(2s+n)(n-1)}{2n}$$

from 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 $a_1, a_2, ..., a_k$, and $1 \leq a_i \leq n \times l$. Different batches are identically independent, so w.l.o.g., suppose $a_1 \geq a_2 \geq ... \geq a_k$. Then the working state is a tuple $(a_1, a_2, ..., a_k)$. 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 $m_1, m_2, ..., m_\infty$, where m_i is the ith 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 $s = (a_1, a_2, ..., a_k)$. 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 $S_a = (a_1, a_2, ..., a_k)$ to another state $S_b = (b_1, b_2, ..., b_k)$. The indices in S_a are sorted by $a_1 \geq a_2 \geq ... \geq a_k$. 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. □

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 $j_1, j_2, ..., j_\infty$ 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'[3]. 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. □

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.

[3]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
$ctog\_bdw$	CPU to GPU bandwidth
$gtoc\_bdw$	GPU to CPU bandwidth
$dtoc\_bdw$	disk to CPU bandwidth
$ctod\_bdw$	CPU to disk bandwidth
mm_flops	GPU flops per second for matrix multiplication
bmm_flops	GPU flops per second for batched matrix multiplication
cpu_flops	CPU flops per second
wg	percentage of weights on GPU
wc	percentage of weights on CPU
wd	percentage of weights on disk
cg	percentage of KV cache on GPU
cc	percentage of KV cache on CPU
cd	percentage of KV cache on disk
hg	percentage of activations on GPU
hc	percentage of activations on CPU
hd	percentage of activations on disk

Table 7. Notation Variables

Objective

$$\text{Minimize} \quad T/bls$$

Then the following constraints describe the calculation of total latency:

$$T = Tpre \cdot l + Tgen \cdot (n-1) \cdot l$$

$$Tpre = \max(ctog^p, gtoc^p, dtoc^p, ctod^p, comp^p)$$

$$\begin{aligned} ctog^p &= \frac{weights\_ctog^p + act\_ctog^p}{ctog\_bdw} \\ &= \frac{1}{ctog\_bdw}((wc+wd)(8h_1^2 + 4h_1 \cdot h_2) \\ &\quad + 2(hc+hd)s \cdot h_1 \cdot bls) \end{aligned}$$

$$\begin{aligned} gtoc^p &= \frac{cache\_gtoc^p + act\_gtoc^p}{gtoc\_bdw} \\ &= \frac{1}{gtoc\_bdw}(4(cc+cd)(s+1)h_1 \cdot bls \\ &\quad + 2(hc+hd)s \cdot h_1 \cdot bls) \end{aligned}$$

$$\begin{aligned} dtoc^p &= \frac{weights\_dtoc^p + act\_dtoc^p}{dtoc\_bdw} \\ &= \frac{1}{dtoc\_bdw}(wd(8h_1^2 + 4h_1 \cdot h_2) \\ &\quad + 2hd \cdot s \cdot h_1 \cdot bls) \end{aligned}$$

$$\begin{aligned} ctod^p &= \frac{cache\_ctod^p + act\_ctod^p}{ctod\_bdw} \\ &= \frac{1}{ctod\_bdw}(4cd \cdot bls \cdot (s+1) \cdot h_1 \\ &\quad + 2hd \cdot s \cdot h_1 \cdot bls) \end{aligned}$$

$$\begin{aligned} comp^p &= \frac{linear\_layer^p}{mm\_flops} + \frac{att^p}{bmm\_flops} \\ &= \frac{bls(8s \cdot h_1^2 + 4s \cdot h_1 \cdot h_2)}{mm\_flops} \\ &\quad + \frac{4bls \cdot s^2 \cdot h_1}{bmm\_flops} \end{aligned}$$

$$Tgen = \max(ctog^g, gtoc^g, dtoc^g, ctod^g, comp^g)$$

$$\begin{aligned} ctog^g &= \frac{weights\_ctog^g + act\_ctog^g}{ctog\_bdw} \\ &= \frac{1}{ctog\_bdw}((wc+wd)(8h_1^2 + 4h_1 \cdot h_2) \\ &\quad + 2(hc+hd)h_1 \cdot bls) \end{aligned}$$

$$\begin{aligned} gtoc^g &= \frac{act\_gtoc^g}{gtoc\_bdw} \\ &= \frac{1}{gtoc\_bdw}(2(hc+hd) \cdot h_1 \cdot bls) \end{aligned}$$

$$\begin{aligned} dtoc^g &= \frac{cache\_dtoc^g + weights\_dtoc^g + act\_dtoc^g}{dtoc\_bdw} \\ &= \frac{1}{dtoc\_bdw}(4cd \cdot bls \cdot (s+n/2) \cdot h_1 \\ &\quad + wd(8h_1^2 + 4h_1 \cdot h_2) \\ &\quad + 2hd \cdot h_1 \cdot bls) \end{aligned}$$

$$\begin{aligned} ctod^g &= \frac{cache\_ctod^g + act\_ctod^g}{ctod\_bdw} \\ &= \frac{1}{ctod\_bdw}(4cd \cdot bls \cdot h_1 + 2hd \cdot h_1 \cdot bls) \end{aligned}$$

$$comp^g = gpu\_comp^g + cpu\_comp^g$$

$$\begin{aligned} gpu\_comp^g &= \frac{linear\_layer^g}{mm\_flops} + \frac{att^g}{bmm\_flops} \\ &= \frac{bls(8h_1^2 + 4h_1 \cdot h_2)}{mm\_flops} \\ &\quad + \frac{4cg \cdot bls \cdot (s+n/2) \cdot h_1}{bmm\_flops} \end{aligned}$$

$$\begin{aligned} cpu\_comp^g &= \frac{att^g}{cpu\_flops} \\ &= \frac{4(cc+cd)bls \cdot (s+n/2) \cdot h_1}{cpu\_flops} \end{aligned}$$

Peak Memory Constraints

- GPU peak memory constraints during prefill:
  GPU memory used to hold a fixed percentage of weights, activations, and cache is

$$\begin{aligned} gpu\_home^p = {} & wg \cdot (8h_1^2 + 4h_1 \cdot h_2) \cdot l \\ & + hg \cdot 2s \cdot h_1 \cdot bls \\ & + 4(s+n)h_1 \cdot cg \cdot bls \cdot l. \end{aligned}$$

  GPU working memory (omit mask):

$$\begin{aligned} qkv^p &= gbs \cdot (2s \cdot h_1 + 3(2s \cdot h_1)) \\ &= gbs \cdot 8s \cdot h_1 \\ att_1^p &= cg \cdot gbs \cdot (2s \cdot h_1 + 2s \cdot h_1 + 2nh \cdot s^2) \\ att_2^p &= cg \cdot gbs \cdot (2nh \cdot s^2 + 2s \cdot h_1 + 2s \cdot h_1) \\ embed^p &= gbs \cdot (2s \cdot h_1 + 2s \cdot h_1) \\ &= gbs \cdot 4s \cdot h_1 \\ mlp_1^p &= gbs \cdot 2(s \cdot h_1 + s \cdot h_2) \\ &= 2gbs \cdot s(h_1 + h_2) \\ mlp_2^p &= gbs \cdot 2(s \cdot h_2 + s \cdot h_1) \\ &= 2gbs \cdot s(h_1 + h_2) \end{aligned}$$

$$\begin{aligned} gpu\_w^p = {} & 2(1 - wg)(8h_1^2 + 4h_1 \cdot h_2) \\ & + (1 - hg) \cdot 2s \cdot h_1 \cdot gbs \\ & + \max(qkv, att_1, att_2, embed, mlp_1, mlp_2) \end{aligned}$$

$$gpu\_peak^p = gpu\_home^p + gpu\_w^p < gmem$$

- GPU peak memory constraints after prefill:
  GPU memory used to hold a fixed percentage of weights, activations, and cache is

$$\begin{aligned} gpu\_home^g = {} & wg \cdot (8h_1^2 + 4h_1 \cdot h_2) \cdot l \\ & + hg \cdot 2h_1 \cdot bls \\ & + 4(s+n)h_1 \cdot cg \cdot bls \cdot l. \end{aligned}$$

  GPU working memory (omit mask):

$$\begin{aligned} qkv^g &= gbs \cdot (2h_1 + 3(2h_1)) = 8gbs \cdot h_1 \\ att_1^g &= cg \cdot gbs \cdot (2h_1 + 2(s+n)h_1 \\ &\qquad + 2nh(s+n)) \\ att_2^g &= cg \cdot gbs \cdot (2nh(s+n) + 2(s+n)h_1 \\ &\qquad + 2h_1) \\ embed^g &= gbs \cdot (2h_1 + 2h_1) = 4gbs \cdot h_1 \\ mlp_1^g &= 2gbs \cdot (h_1 + h_2) \\ mlp_2^g &= 2gbs \cdot (h_2 + h_1) \end{aligned}$$

$$\begin{aligned} gpu\_w^g = {} & 2(1 - wg)(8h_1^2 + 4h_1 \cdot h_2) \\ & + (1 - hg) \cdot 2s \cdot h_1 \cdot gbs \\ & + \max(qkv^g, att_1^g, att_2^g, embed^g, mlp_1^g, mlp_2^g) \end{aligned}$$

$$gpu\_peak^g = gpu\_home^g + gpu\_w^g < gmem$$

- CPU peak memory constraints during prefill:
  CPU memory used to hold a fixed percentage of weights, activations, and cache is

$$\begin{aligned} cpu\_home^p = {} & wc \cdot (8h_1^2 + 4h_1 \cdot h_2) \cdot l \\ & + hc \cdot 2s \cdot h_1 \cdot bls \\ & + 4(s+n)h_1 \cdot cc \cdot bls \cdot l. \end{aligned}$$

  CPU working memory:

$$\begin{aligned} cpu\_w^p = {} & (1 - wg)(8h_1^2 + 4h_1 \cdot h_2) \\ & + (1 - hg) \cdot 2s \cdot h_1 \cdot gbs. \end{aligned}$$

$$cpu\_peak^p = cpu\_home^p + cpu\_w^p < cmem$$

- CPU peak memory constraints after prefill:
  CPU memory used to hold fixed percentage of weights, activations, and cache is

$$\begin{aligned} cpu\_home^g = {} & wc \cdot (8h_1^2 + 4h_1 \cdot h_2) \cdot l \\ & + hc \cdot 2h_1 \cdot bls \\ & + 4(s+n)h_1 \cdot cc \cdot bls \cdot l. \end{aligned}$$

  CPU working memory:

$$\begin{aligned} cpu\_w^g = {} & wd(8h_1^2 + 4h_1 \cdot h_2) \\ & + 2hd \cdot 2 \cdot h_1 \cdot gbs \\ & + 2cd \cdot 4(s+n)h_1 \cdot gbs \\ & + 2nh \cdot (s+n) \cdot gbs \\ & + 2h_1 \cdot gbs. \end{aligned}$$

$$cpu\_peak^g = cpu\_home^g + cpu\_w^g < cmem$$

- NVMe peak memory constraints:

$$\begin{aligned} nvme\_peak = {} & (8h_1^2 + 4h_1 \cdot h_2) \cdot wd \cdot l \\ & + hd \cdot 2s \cdot h_1 \cdot bls \\ & + cd \cdot 4(s+n)h_1 \cdot bls \cdot l \\ < {} & nmem \end{aligned}$$

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.

Abalation 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. Comparing 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 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

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 baching 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, wg, wc, cg, cc, hg, hc).

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, wg, wc, cg, cc, hg, hc).

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, 100, 0)	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, wg, wc, cg, cc, hg, hc).

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 × #GPU-batch, wg, wc, cg, cc, hg, hc).

Seq. length	128 + 128		
Model size	6.7B	30B	175B
Accelerate	73.411 (5×1, 100, 0, 100, 0, 100, 0)	1.547 (16×1, 0, 100, 100, 0, 100, 0)	0.021 (4×1, 0, 0, 100, 0, 100, 0)
DeepSpeed	19.193 (36×1, 0, 100, 100, 0, 100, 0)	1.717 (12×1, 0, 100, 100, 0, 100, 0)	0.024 (3×1, 0, 0, 100, 0, 100, 0)
FlexGen	106.404 (7×1, 100, 0, 100, 0, 100, 0)	24.634 (32×10, 25, 75, 0, 100, 100, 0)	2.409 (64×8, 0, 50, 0, 0, 0, 100)
FlexGen (c)	92.568 (196×1, 100, 0, 100, 0, 100, 0)	39.141 (128×8, 25, 75, 0, 100, 0, 100)	4.264 (80×3, 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 × #GPU-batch, wg, wc, cg, cc, hg, hc).

Seq. length	512 + 8		
Model size	6.7B	30B	175B
Accelerate	17.290 (2×1, 100, 0, 100, 0, 100, 0)	0.628 (7×1, 0, 100, 100, 0, 100, 0)	0.009 (2×1, 0, 0, 100, 0, 100, 0)
DeepSpeed	9.055 (18×1, 0, 100, 100, 0, 100, 0)	0.872 (6×1, 0, 100, 100, 0, 100, 0)	0.007 (1×1, 0, 0, 100, 0, 100, 0)
FlexGen	16.425 (2×1, 100, 0, 100, 0, 100, 0)	3.938 (512×8, 20, 80, 0, 100, 0, 100)	0.451 (32×8, 0, 50, 0, 0, 0, 100)
FlexGen (c)	14.244 (76×1, 100, 0, 100, 0, 100, 0)	4.019 (16×36, 25, 75, 0, 100, 0, 100)	0.559 (48×3, 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 / **612** (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)	**1.122** / 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 / **153** (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)
-	-	-	**8.70** / 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-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 throughput. (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 (token/s)	175B (token/s)
48	3	20	80	0	100	0	100	**7.32**	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	**0.49**
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	**160**	OOM
1	1	0	50	100	0	100	0	737	**3107**
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 (48×3, 20, 80, 0, 100, 0, 100)	0.69 (32×8, 0, 50, 0, 0, 0, 100)
No policy search	7.26 (48×3, 0, 100, 0, 100, 0, 100)	0.27 (32×1, 0, 50, 0, 0, 0, 100)
No overlapping	5.86 (48×3, 20, 80, 0, 100, 0, 100)	0.59 (32×8, 0, 50, 0, 0, 0, 100)
No CPU compute	4.03 (48×3, 20, 80, 0, 100, 0, 100)	0.62 (32×8, 0, 50, 0, 0, 0, 100)
No disk	7.32 (48×3, 20, 80, 0, 100, 0, 100)	OOM
w/ DeepSpeed policy	1.57 (8×1, 0, 100, 100, 0, 100, 0)	0.01 (2×1, 0, 0, 100, 0, 100, 0)

Table 24. Generation throughput (token/s) on hardware specifed 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%

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 Mellempudi, Marius Cornea, Alexander Heinecke, Pradeep Dubey
Intel
{naveen.k.mellempudi, 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 representatio 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 corner stoen 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], posits, 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. Mellempudi 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	$S.1111.111_2$	$S.11111.\{01, 10, 11\}_2$
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	$S.0000.001_2 = 2^{-9}$	$S.00000.01_2 = 2^{-16}$

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 extents 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 Trasnformer-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 MobilNet v2.

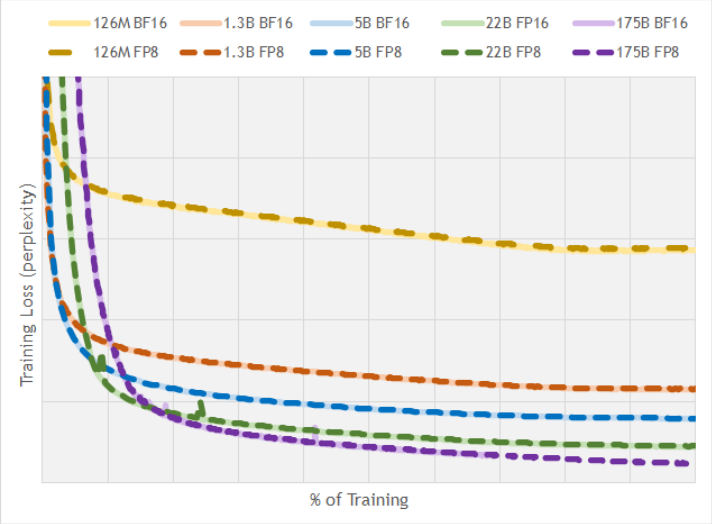


Figure 1: Training loss (perplexity) curves for various GPT-3 models. x-axis is normalized number of iterations.

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 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

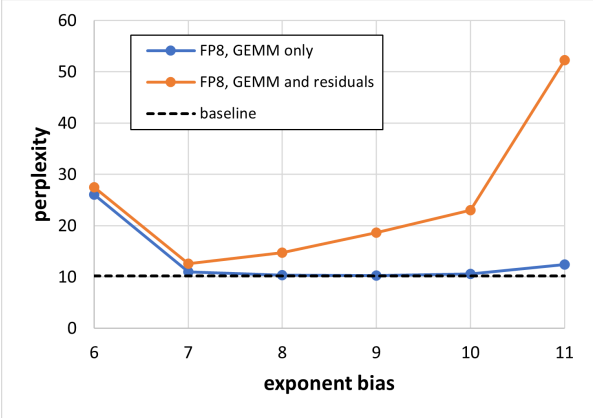


Figure 2: 1.3B GPT3 perplexity when bfloat16-trained model weight and activation input tensors are cast to E4M3 format with various exponent biases, no per-tensor scaling.

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	**88.09**
BERT Large	SQuAD v1.1 (F1)	90.87	89.65	**90.94**
GPT3 126M	wikitext103 (perplexity)	19.01	28.37	**19.43**
GPT3 1.3B	wikitext103 (perplexity)	10.19	12.74	**10.29**
GPT3 6.7B	wikitext103 (perplexity)	8.51	10.29	**8.41**

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 Anbudurai, Vandana Balan, Harsha Bojja, Joe Boyd, Matthew Breitbach, Claudio Caldato, Anna Calvo, Garret Catron, Sneh Chandwani, Panos Christeas, Brad Cottel, Brian Coutinho, Arun Dalli, Abhishek Dhanotia, Oniel Duncan, Roman Dzhabarov, Simon Elmir, 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 Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek 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 Mellempudi, 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 Herve 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, Shuohui 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.

LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale

Tim Dettmers[λ*] **Mike Lewis**[†] **Younes Belkada**[§∓] **Luke Zettlemoyer**[†λ]

University of Washington[λ]
Facebook AI Research[†]
Hugging Face[§]
ENS Paris-Saclay[∓]

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%[2] 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.
[2]Other parameters come mostly from the embedding layer. A tiny amount comes from norms and biases.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

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 denormalizing by the outer product of column and row normalization constants before we perform the next operation.

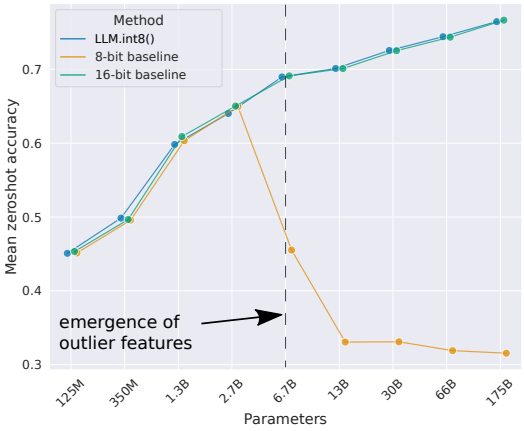


Figure 1: OPT model mean zeroshot accuracy for WinoGrande, HellaSwag, PIQA, and LAMBADA datasets. Shown is the 16-bit baseline, the most precise previous 8-bit quantization method as a baseline, and our new 8-bit quantization method, LLM.int8(). We can see once systematic outliers occur at a scale of 6.7B parameters, regular quantization methods fail, while LLM.int8() maintains 16-bit accuracy.

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[3] 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.

[3] `https://github.com/TimDettmers/bitsandbytes`

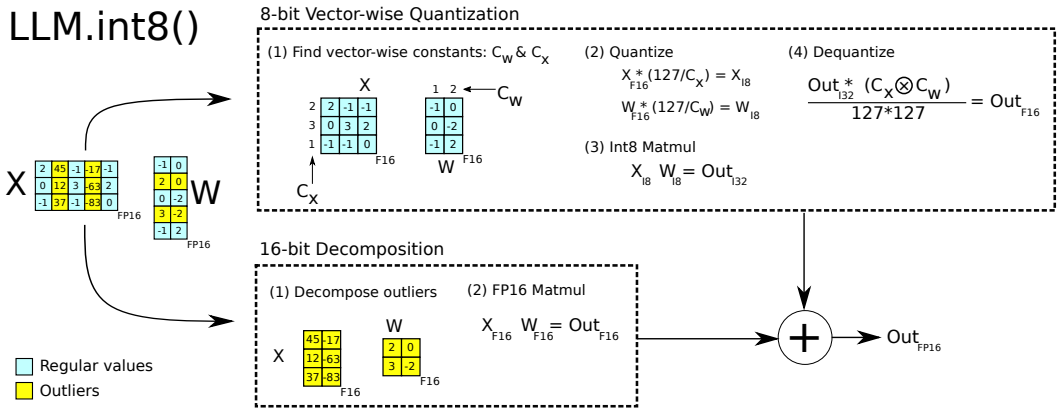


Figure 2: Schematic of LLM.int8(). Given 16-bit floating-point inputs $\mathbf{X}_{f16}$ and weights $\mathbf{W}_{f16}$, the features and weights are decomposed into sub-matrices of large magnitude features and other values. The outlier feature matrices are multiplied in 16-bit. All other values are 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 dequantization 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 related 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:

$$\mathbf{X}_{i8} = \left\lfloor \frac{127 \cdot \mathbf{X}_{f16}}{\max_{ij}(|\mathbf{X}_{f16_{ij}}|)} \right\rceil = \left\lfloor \frac{127}{\|\mathbf{X}_{f16}\|_\infty} \mathbf{X}_{f16} \right\rceil = \left\lfloor s_{x_{f16}} \mathbf{X}_{f16} \right\rceil,$$

where $\lfloor \rceil$ 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:

$$nd_{x_{f16}} = \frac{2 \cdot 127}{\max_{ij}(\mathbf{X}^{ij}_{f16}) - \min_{ij}(\mathbf{X}^{ij}_{f16})} \tag{1}$$

$$zp_{x_{i16}} = \left\lfloor \mathbf{X}_{f16} \cdot \min_{ij}(\mathbf{X}^{ij}_{f16}) \right\rceil \tag{2}$$

$$\mathbf{X}_{i8} = \left\lfloor nd_{x_{f16}} \mathbf{X}_{f16} \right\rceil \tag{3}$$

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[4] 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:

$$C_{i32} = \text{multiply}_{i16}(A_{zp_{a_{i16}}}, B_{zp_{b_{i16}}}) = (A_{i8} + zp_{a_{i16}})(B_{i8} + zp_{b_{i16}}) \tag{4}$$

where unrolling is required if the instruction multiply_{i16} is not available such as on GPUs or TPUs:

$$C_{i32} = A_{i8}B_{i8} + A_{i8}zp_{b_{i16}} + B_{i8}zp_{a_{i16}} + zp_{a_{i16}}zp_{b_{i16}}, \tag{5}$$

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 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:

$$\begin{aligned} \mathbf{X}_{f16}\mathbf{W}_{f16} = \mathbf{C}_{f16} &\approx \frac{1}{c_{x_{f16}}c_{w_{f16}}}\mathbf{C}_{i32} = S_{f16} \cdot \mathbf{C}_{i32} \\ &\approx S_{f16} \cdot \mathbf{A}_{i8}\mathbf{B}_{i8} = S_{f16} \cdot Q(\mathbf{A}_{f16})\, Q(\mathbf{B}_{f16}), \end{aligned} \tag{6}$$

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 (≈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:

$$\mathbf{C}_{f_{16}} \approx \frac{1}{\mathbf{c}_{x_{f16}} \otimes \mathbf{c}_{w_{f16}}}\mathbf{C}_{i32} = S \cdot \mathbf{C}_{i32} = \mathbf{S} \cdot \mathbf{A}_{i8}\mathbf{B}_{i8} = \mathbf{S} \cdot Q(\mathbf{A}_{f16})\, Q(\mathbf{B}_{f16}), \tag{7}$$

which we term *vector-wise quantization* for matrix multiplication.

[4] `https://www.felixcloutier.com/x86/pmaddubsw`

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 α. 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:

$$\mathbf{C}_{f16} \approx \sum_{h \in O} \mathbf{X}^h_{f16} \mathbf{W}^h_{f16} + \mathbf{S}_{f16} \cdot \sum_{h \notin O} \mathbf{X}^h_{i8} \mathbf{W}^h_{i8} \tag{8}$$

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.[5] 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.

[5] `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	**13.24**	**12.45**
Zeropoint LLM.int8() (vector-wise + decomp)	**25.69**	**15.92**	**14.43**	**13.24**	**12.45**

When we look at the scaling trends of zeroshot 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

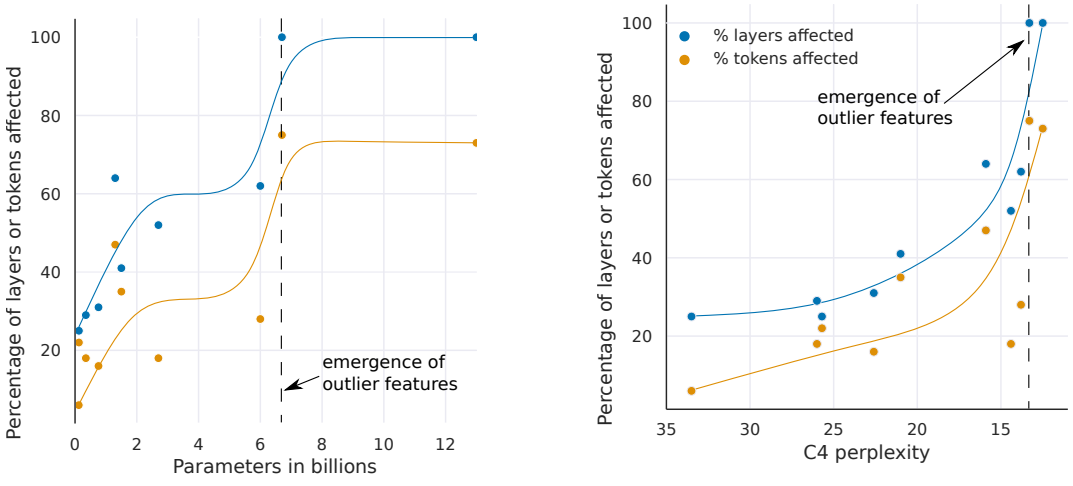
As we scale transformers, outlier features with large magnitudes emerge and strongly affect *all* layers and their quantization. Given a hidden state $\mathbf{X} \in \mathbb{R}^{s \times h}$ where s is the sequence/token dimension and h the hidden/feature dimension, we define a feature to be a particular 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 to 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 $\mathbf{X}_l \in \mathbb{R}^{s \times h}, l = 0...L$ 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 $0...L$ 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



(a) (b)

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.

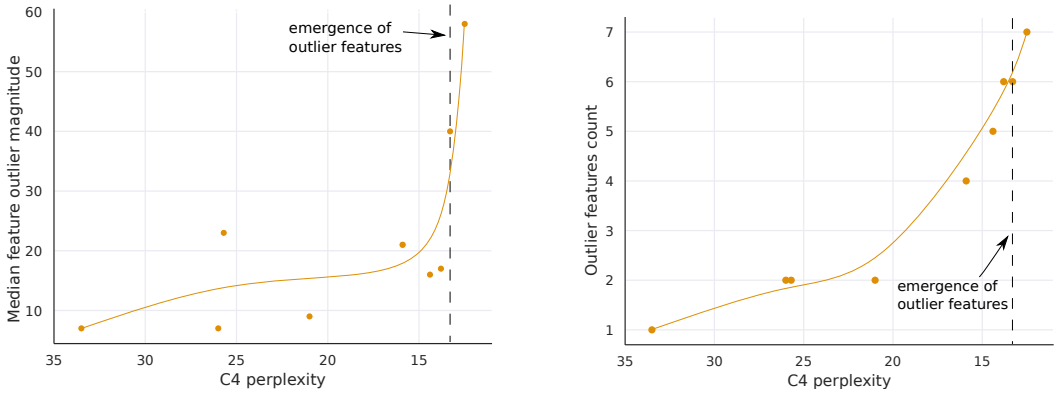
Our reasoning for these thresholds is as follows. We find that using mixed-precision decomposition, perplexity degradation stops if we treat any feature with a magnitude 6 or larger as an outlier feature. For the number of layers affected by outliers, we find that outlier features are *systematic* in 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.



(a) (b)

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; Mellempudi 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.

			Largest Model that can be run	
Class	Hardware	GPU Memory	8-bit	16-bit
Enterprise	8x A100	80 GB	**OPT-175B / BLOOM**	**OPT-175B / BLOOM**
Enterprise	8x A100	40 GB	**OPT-175B / BLOOM**	OPT-66B
Academic server	8x RTX 3090	24 GB	**OPT-175B / BLOOM**	OPT-66B
Academic desktop	4x RTX 3090	24 GB	**OPT-66B**	OPT-30B
Paid Cloud	Colab Pro	15 GB	**OPT-13B**	GPT-J-6B
Free Cloud	Colab	12 GB	**T0/T5-11B**	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 Köller 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.

Zafrir, 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 exper2imental 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.

			Largest Model that can be run	
Class	Hardware	GPU Memory	8-bit	16-bit
Enterprise	8x A100	80 GB	**OPT-175B / BLOOM**	**OPT-175B / BLOOM**
Enterprise	8x A100	40 GB	**OPT-175B / BLOOM**	OPT-66B
Academic server	8x RTX 3090	24 GB	**OPT-175B / BLOOM**	OPT-66B
Academic desktop	4x RTX 3090	24 GB	**OPT-66B**	OPT-30B
Paid Cloud	Colab Pro	15 GB	**OPT-13B**	GPT-J-6B
Free Cloud	Colab	12 GB	**T0/T5-11B**	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 (Zafrir 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 absmax 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).

			Outliers		Frequency			Top-1 softmax p	
Model	PPL↓	Params	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	Small	Medium	Large	XL	2.7B	6.7B	13B	175B
Model dimension	768	1024	1536	2048	2560	4096	5140	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	**0.43x**	**0.49x**	**0.74x**	**0.91x**	**0.94x**	**1.18x**	**1.59x**	**2.00x**
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	**239**	**32**	9.94
LLM.int8()	8xA100 80GB	253	34	10.44
LLM.int8()	4xA100 80GB	246	33	9.40
LLM.int8()	3xA100 80GB	247	33	**9.11**

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 iteself 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.

	Is 8-bit				
Params	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 comparel 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 (Zafrir 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	**86.8**	96.2	90.4	67.5	91.9	88.65
Vector-wise	**90.2**	**94.7**	**92.3**	85.4	**96.4**	**91.0**	**68.6**	**91.9**	**88.81**

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 multplication. 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										
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

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 4×4 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 4×4 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 256×256 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 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× 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.

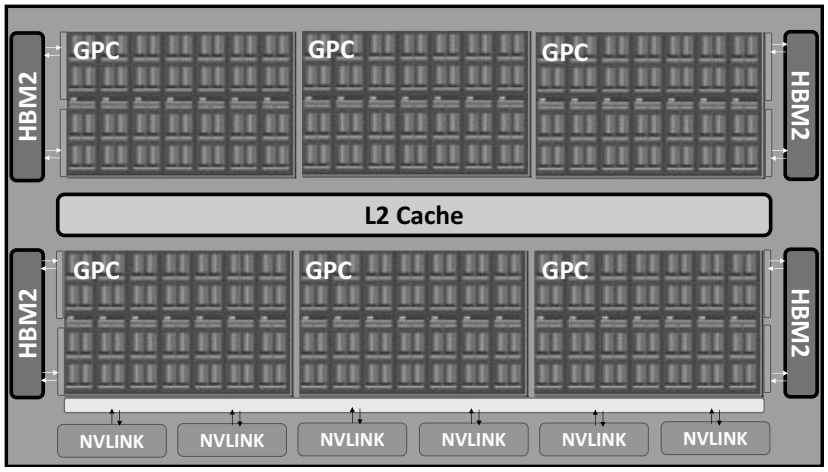


Fig. 1: Volta GV100 GPU architecture features six GPCs and 16 GB HBM2. Adapted from [18].

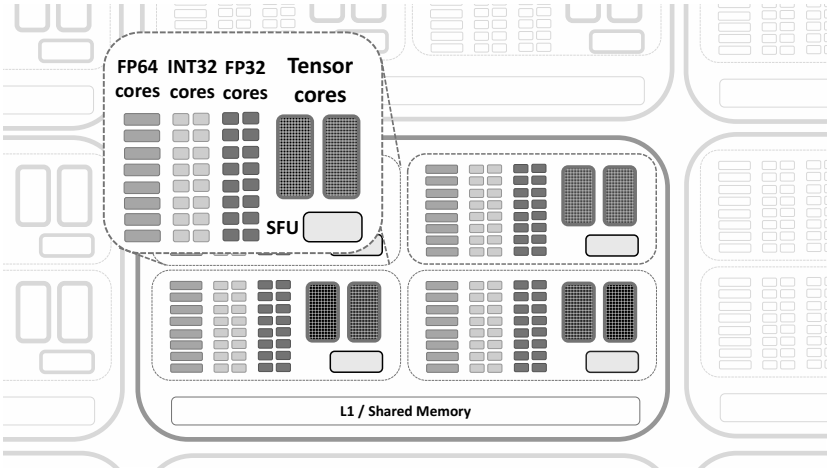


Fig. 2: Simplified diagram of the Volta SM architecture. The NVIDIA Tesla V100 uses 80 SMs.

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 4×4 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.

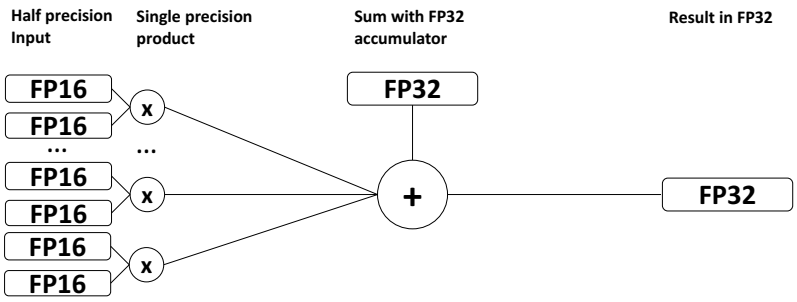


Fig. 3: FMAs in NVIDIA Tensor Cores.

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 16×16 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 4×4 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 16x16 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 16×16 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. 16×16, 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[1]. 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 NVIDA 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.

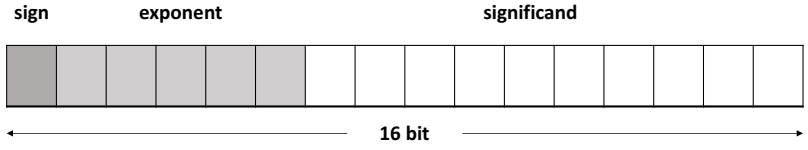


Fig. 4: Half precision floating-point number representation.

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.

[1]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):

$$R_A = A_{single} - A_{half}. \tag{1}$$

We manipulate $A_{single}B_{half}$ and compute on Tensor Cores using the distributive property of matrix multiplication and sum as

$$A_{single}B_{half} = (A_{single} - A_{half} + A_{half})B_{half} = (R_A + A_{half})B_{half} = R_AB_{half} + A_{half}B_{half}. \tag{2}$$

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:

$$A_{single}B_{single} = (R_A + A_{half})(R_B + B_{half}) = R_AR_B + A_{half}R_B + R_AB_{half} + A_{half}B_{half}. \tag{3}$$

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 [1], we wrote a simple implementation for testing purposes, extending the code in Listing 1. For batched `GEMM`, we only use square 16×16 matrices. In this case, the CUDA execution configuration consists of 512 threads per block. Since a 16×16 matrix multiplication is executed by one Warp (32 threads), 16 matrix multiplications are executed per thread block.

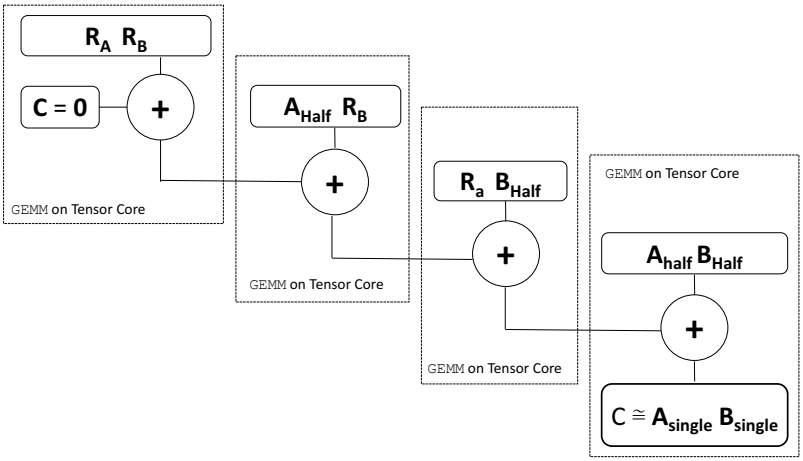


Fig. 5: Implementation of precision refinement using four pipelined GEMM on Tensor Cores.

To quantify the precision loss due to mixed precision computations, we first calculate the error matrix e as $e = (C_{half} - C_{single})$, where C_{half} is the result of the matrix multiplication with half precision input and C_{single} is the result with single precision input. We then apply the max norm $\|e\|_{Max} = \max(|e_{i,j}|)$. 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× and 3× 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 16×16 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 16×16 matrix multiplies increases the performance of the GEMM with and without Tensor Cores. When using cuBLAS batched `sgemm` for $batchsize > 131,072$,

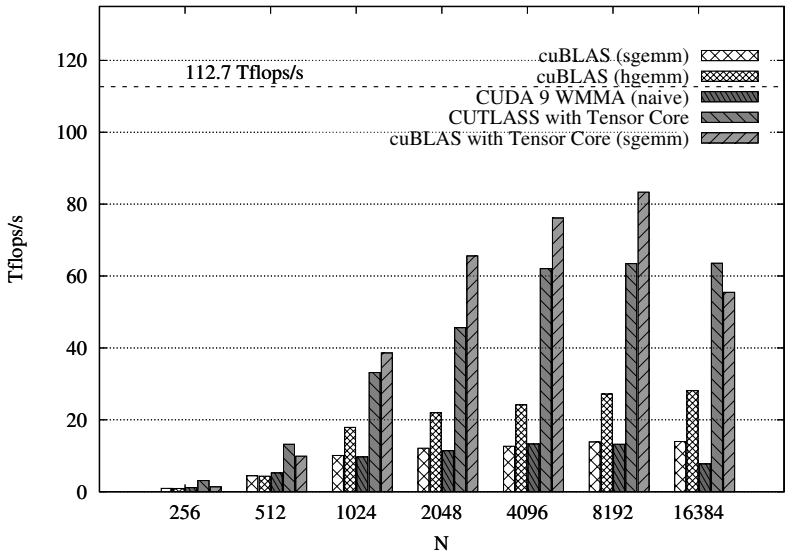


Fig. 6: GEMM performance without Tensor Cores in single and half precision (white bars) and with Tensor Cores using naive implementation with CUDA 9 WMMA, CUTLASS and cuBLAS (grey bars) varying with matrix size N.

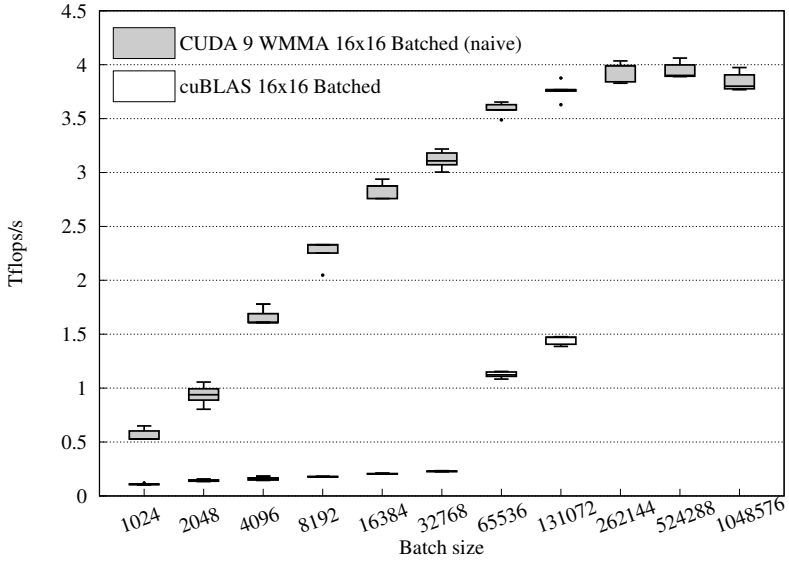


Fig. 7: Performance of cuBLAS batched sgemm on CUDA cores, and CUDA 9 WMMA implementation performing *batch size* 16×16 matrix multiplies. The cuBLAS batched sgemm cannot run for more than 131,072 multiplications as they require more memory than the available one on the Tesla V100 GPU.

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× and 12× 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 grey 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.

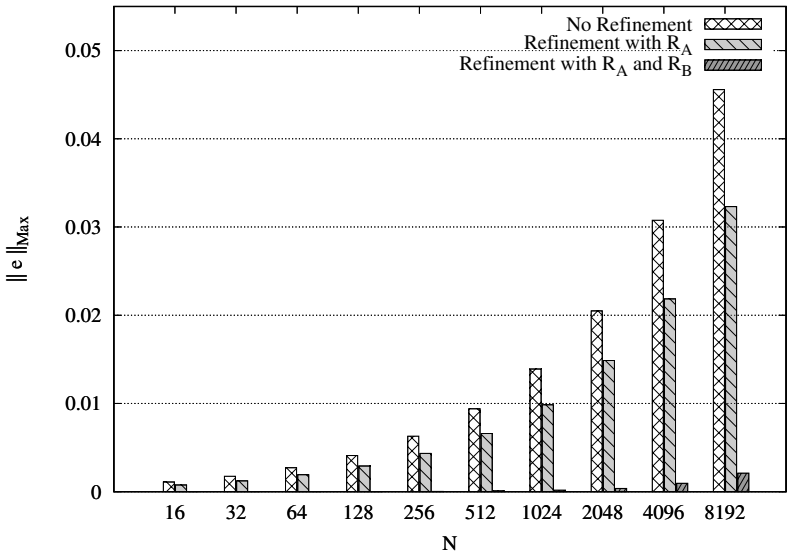


Fig. 8: Error in half precision (no refinement, white bars), using precision refinement with R_A, and precision refinement with both R_A and R_B varying the matrix size N.

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 ± 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.

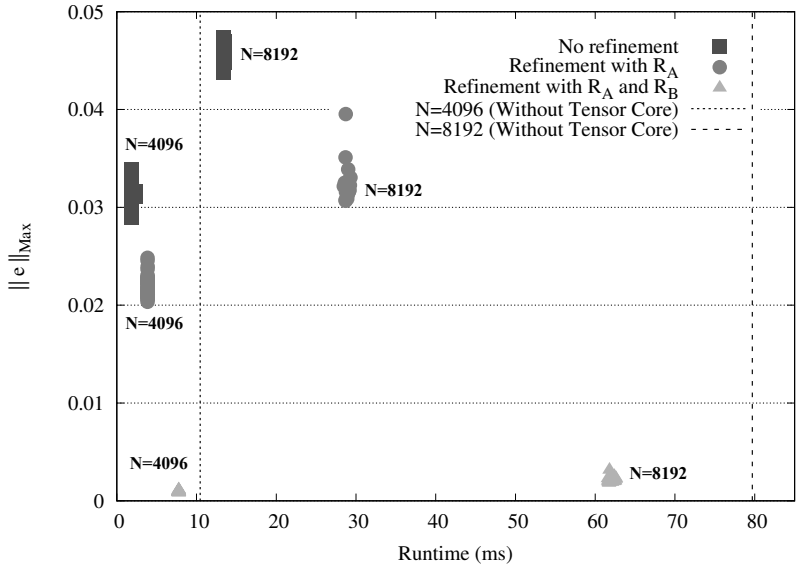


Fig. 9: Scatter plot with $\|e\|_{Max}$ on x axis and runtime on the y axis for `GEMM` with no refinement (squares), refinement with only R_A (circles), and with both R_A and R_B (triangles) for $N = 8,192$ and $N = 4,096$. The two dashed lines represent the execution time for `sgemm` without Tensor 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× 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× - 12× 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. Hornof, 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, 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. Heinecke, 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

Natalia Gimelshein
NVIDIA
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 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.

Preprint. Work in progress.

2 Original softmax

Function $y = Softmax(x)$ is defined as:

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}} \tag{1}$$

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: $\quad d_j \leftarrow d_{j-1} + e^{x_j}$
4: **end for**
5: **for** $i \leftarrow 1, V$ **do**
6: $\quad y_i \leftarrow \frac{e^{x_i}}{d_V}$
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:

$$y_i = \frac{e^{x_i - \max_{k=1}^{V} x_k}}{\sum_{j=1}^{V} e^{x_j - \max_{k=1}^{V} x_k}} \tag{2}$$

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: $\quad m_k \leftarrow \max(m_{k-1}, x_k)$
4: **end for**
5: $d_0 \leftarrow 0$
6: **for** $j \leftarrow 1, V$ **do**
7: $\quad d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
8: **end for**
9: **for** $i \leftarrow 1, V$ **do**
10: $\quad y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
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: $\quad m_j \leftarrow \max\left(m_{j-1}, x_j\right)$
5: $\quad d_j \leftarrow d_{j-1} \times e^{m_{j-1}-m_j} + e^{x_j-m_j}$
6: **end for**
7: **for** $i \leftarrow 1, V$ **do**
8: $\quad y_i \leftarrow \frac{e^{x_i-m_V}}{d_V}$
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 $V = S$

$$
\begin{aligned}
m_S &\leftarrow \max\left(m_{S-1}, x_S\right) && \text{by line 4 of the algorithm 3}\\
&= \max(\max_{k=1}^{S-1} x_k, x_S) && \text{by the inductive hypothesis}\\
&= \max_{k=1}^{S} x_k\\
d_S &\leftarrow d_{S-1} \times e^{m_{S-1}-m_S} + e^{x_S-m_S} && \text{by line 5 of the algorithm 3}\\
&= \left(\sum_{j=1}^{S-1} e^{x_j-m_{S-1}}\right) \times e^{m_{S-1}-m_S} + e^{x_S-m_S} && \text{by the inductive hypothesis}\\
&= \sum_{j=1}^{S-1} e^{x_j-m_S} + e^{x_S-m_S}\\
&= \sum_{j=1}^{S} e^{x_j-m_S}
\end{aligned}
$$

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:

$$\begin{bmatrix} m_V \\ d_V \end{bmatrix} = \begin{bmatrix} x_1 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} x_2 \\ 1 \end{bmatrix} \oplus ... \oplus \begin{bmatrix} x_V \\ 1 \end{bmatrix} \tag{3}$$

where $x_i, m_V, d_V \in \mathbb{R}$. The binary operation $\oplus : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^2$ is defined as:

$$\begin{bmatrix} m_i \\ d_i \end{bmatrix} \oplus \begin{bmatrix} m_j \\ d_j \end{bmatrix} = \begin{bmatrix} \max(m_i, m_j) \\ d_i \times e^{m_i - \max(m_i, m_j)} + d_j \times e^{m_j - \max(m_i, m_j)} \end{bmatrix} \tag{4}$$

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:

$$TopK(y) = (v, z) : v_i = y_{z_i}, v_i \geq y_j, \forall i \in [1, K], \forall j \notin z \tag{5}$$

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

$m_0 \leftarrow -\infty$
$d_0 \leftarrow 0$
$u \leftarrow \{-\infty, -\infty, \ldots, -\infty\}^T, u \in \mathbb{R}^{K+1}$ ▷ The 1st K elems will hold running TopK values
$p \leftarrow \{-1, -1, \ldots, -1\}^T, p \in \mathbb{Z}^{K+1}$ ▷ ... and their indices
for $j \leftarrow 1, V$ **do**
    $m_j \leftarrow \max(m_{j-1}, x_j)$
    $d_j \leftarrow d_{j-1} \times e^{m_{j-1} - m_j} + e^{x_j - m_j}$
    $u_{K+1} \leftarrow x_j$ ▷ Initialize $K+1$ elem with new value from input vector
    $p_{K+1} \leftarrow j$ ▷ ... and its index
    $k \leftarrow K$ ▷ 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.
    while $k \geq 1$ and $u_k < u_{k+1}$ **do**
        swap(u_k, u_{k+1})
        swap(p_k, p_{k+1})
        $k \leftarrow k - 1$
    end while
end for
for $i \leftarrow 1, K$ **do** ▷ The algorithm stores only K values and their indices
    $v_i \leftarrow \frac{e^{u_i - m_V}}{d_V}$
    $z_i \leftarrow p_i$
end for

ECC on, persistent mode on, CUDA Toolkit 9.1. Source code of the benchmark is available at 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 and the small batch case corresponds to online inference with too few vectors to occupy the device fully.

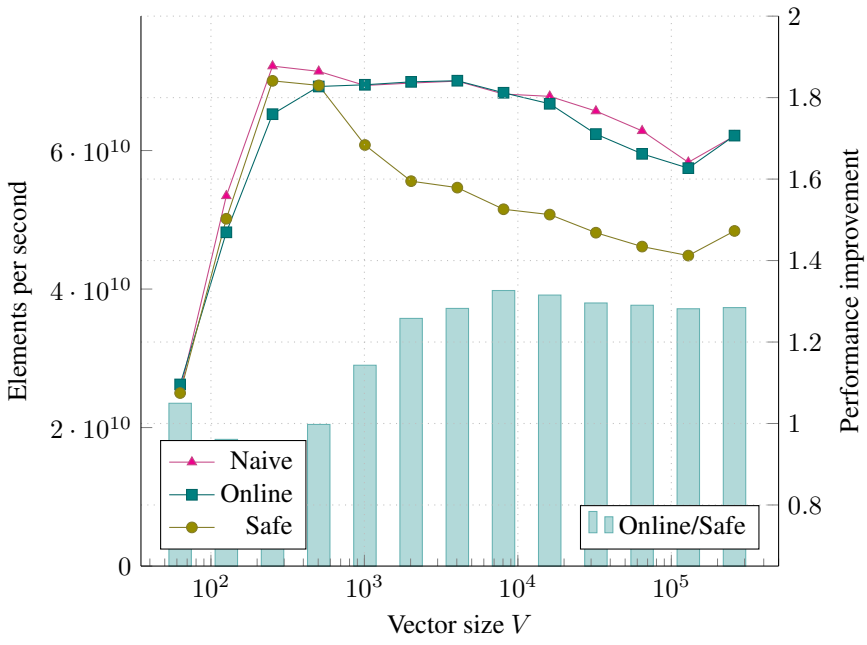


Figure 1: Benchmarking softmax, Tesla V100, fp32, batch size 4000 vectors

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 ~ 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.

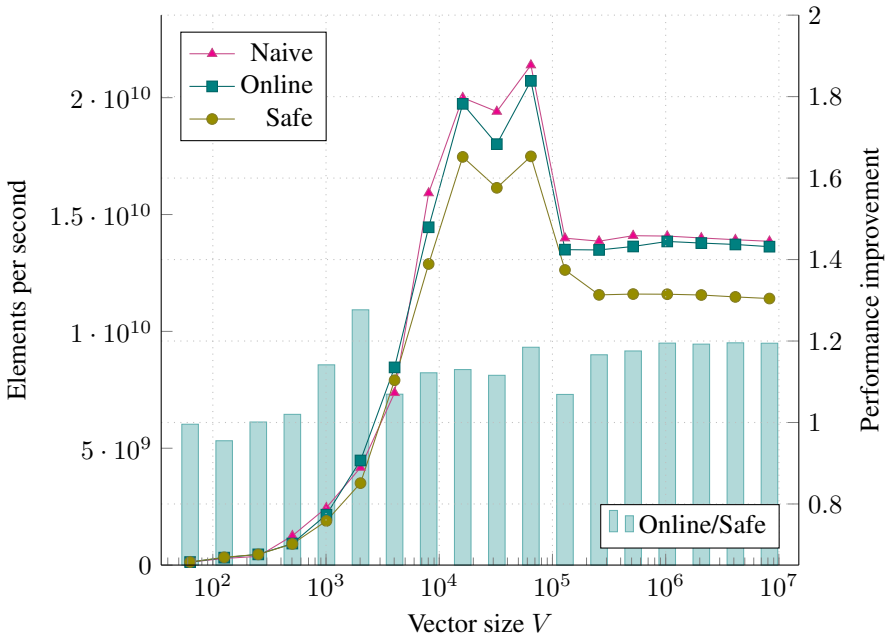


Figure 2: Benchmarking softmax, Tesla V100, fp32, batch size 10 vectors

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 ~ 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 Ks. For these cases the TopK is dominating (in terms of runtime) over the Softmax. Getting rid of separate Softmax and fusing the normalization term

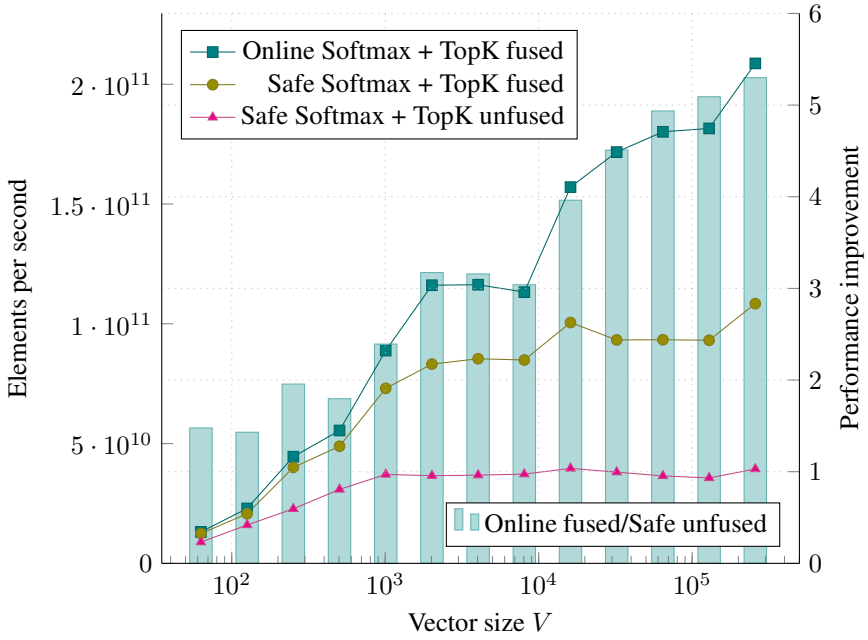


Figure 3: Benchmarking softmax and top-k, Tesla V100, fp32, batch size 4000 vectors

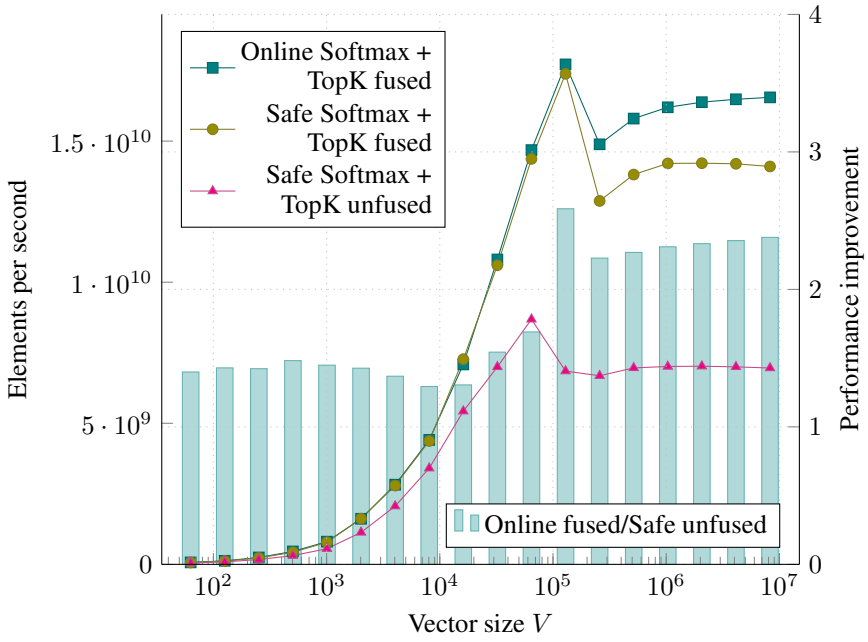


Figure 4: Benchmarking softmax and top-k, Tesla V100, fp32, batch size 10 vectors

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-

[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`.

[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[1,*] Zhuohan Li[1,*] Siyuan Zhuang[1] Ying Sheng[1,2] Lianmin Zheng[1] Cody Hao Yu[3]
Joseph E. Gonzalez[1] Hao Zhang[4] Ion Stoica[1]
[1]UC Berkeley [2]Stanford University [3]Independent Researcher [4]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 the cost per request—of *LLM serving* systems is becoming more important.

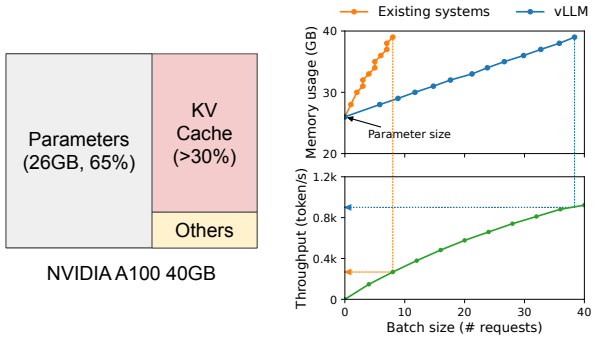


Figure 1. *Left:* Memory layout when serving an LLM with 13B parameters on NVIDIA A100. The parameters (gray) persist in GPU memory throughout serving. The memory for the KV cache (red) is (de)allocated per serving request. A small amount of memory (yellow) is used ephemerally for activation. *Right:* 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.

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

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10.
https://doi.org/10.1145/3600006.3613165

*Equal contribution.

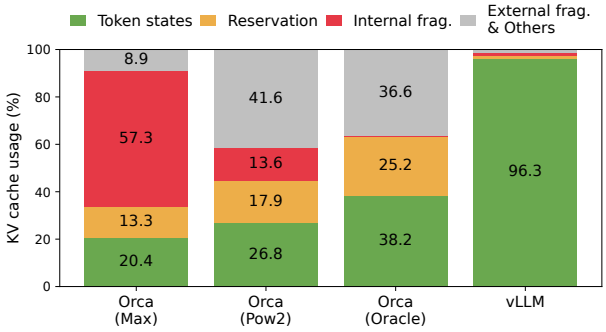


Figure 2. Average percentage of memory wastes in different LLM serving systems during the experiment in §6.2.

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 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 $(x_1, \ldots, x_n)$. 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]):

$$P(x) = P(x_1) \cdot P(x_2 \mid x_1) \cdots P(x_n \mid x_1, \ldots, x_{n-1}). \quad (1)$$

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 $(x_1, \ldots, x_n) \in \mathbb{R}^{n \times d}$, 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 $(x_1, \ldots, x_n)$, and the LLM service generates a list of output tokens $(x_{n+1}, \ldots, x_{n+T})$ 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 $(x_1, \ldots, x_n)$ as input and computes the probability of the first new token $P(x_{n+1} \mid x_1, \ldots, x_n)$. During this process, also generates the key vectors $k_1, \ldots, k_n$ and value vectors $v_1, \ldots, v_n$. Since prompt tokens $x_1, \ldots, x_n$ 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 $P(x_{n+t+1} \mid x_1, \ldots, x_{n+t})$ with the key vectors $k_1, \ldots, k_{n+t}$ and value vectors $v_1, \ldots, v_{n+t}$. 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.

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.

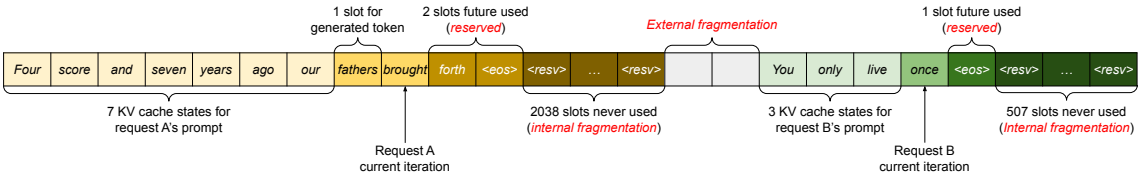


Figure 3. KV cache memory management in existing systems. Three types of memory wastes – reserved, internal fragmentation, and external fragmentation – exist that prevent other requests from fitting into the memory. The token in each memory slot represents its KV cache. Note the same tokens can have 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) × 5120 (hidden state size) × 40 (number of layers) × 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%.

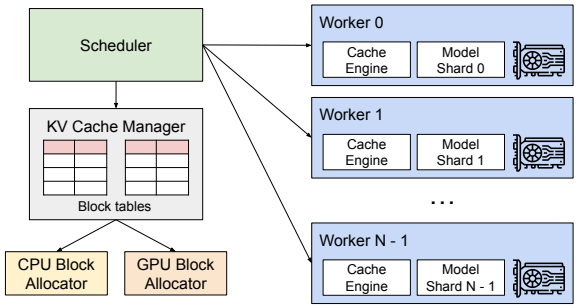


Figure 4. vLLM system overview.

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,[1] which we denote as *KV block size* (B). Denote the key block $K_j = (k_{(j-1)B+1}, \ldots, k_{jB})$ and value block $V_j = (v_{(j-1)B+1}, \ldots, v_{jB})$. The attention computation in Eq. 4 can be transformed into the following block-wise computation:

$$A_{ij} = \frac{\exp(q_i^\top K_j / \sqrt{d})}{\sum_{t=1}^{\lceil i/B \rceil} \exp(q_i^\top K_t \mathbf{1} / \sqrt{d})}, \quad o_i = \sum_{j=1}^{\lceil i/B \rceil} V_j A_{ij}^\top, \tag{4}$$

where $A_{ij} = (a_{i,(j-1)B+1}, \ldots, a_{i,jB})$ is the row vector of attention score on j-th KV block.

[1] 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.

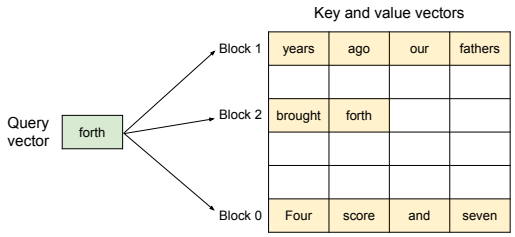


Figure 5. Illustration of the PagedAttention algorithm, where the attention key and values vectors are stored as non-contiguous blocks in the memory.

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

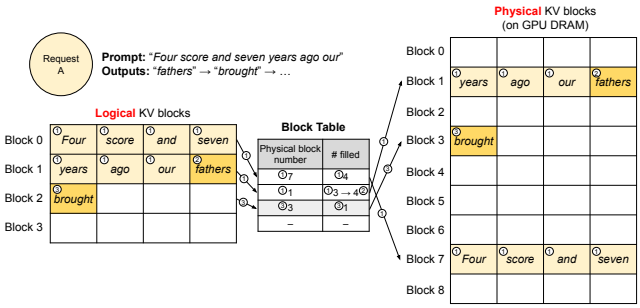


Figure 6. Block table translation in vLLM.

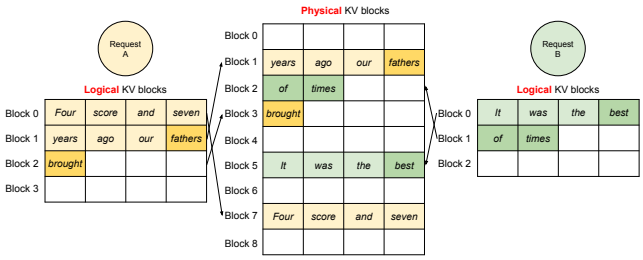


Figure 7. Storing the KV cache of two requests at the same time in vLLM.

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 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

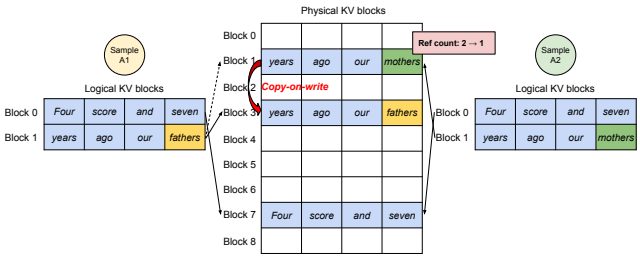


Figure 8. Parallel sampling example.

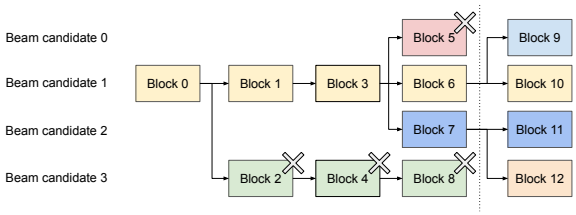


Figure 9. Beam search example.

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 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

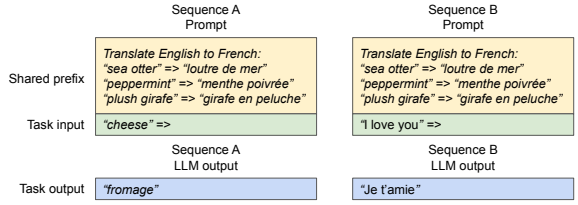


Figure 10. Shared prompt example for machine translation. The examples are adopted from [5].

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? 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

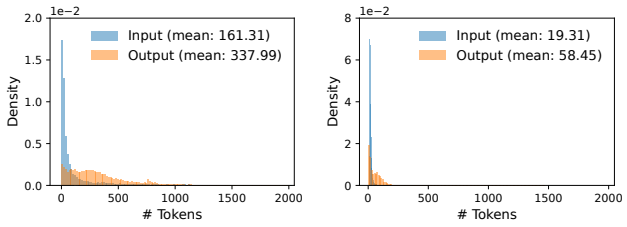
to perform block-wise matrix multiplication, and the the GPUs constantly synchronize intermediate results via an all-reduce operation. Specifically, the attention operator is split on the attention head dimension, each SPMD process takes care of a subset of attention heads in multi-head attention.

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



(a) ShareGPT **(b)** Alpaca

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.

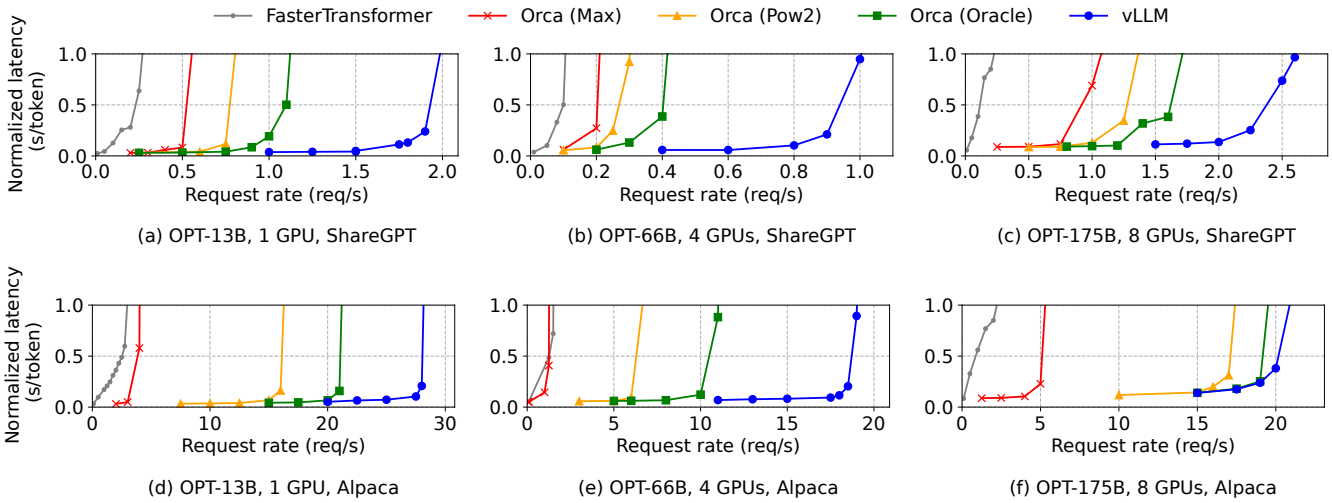
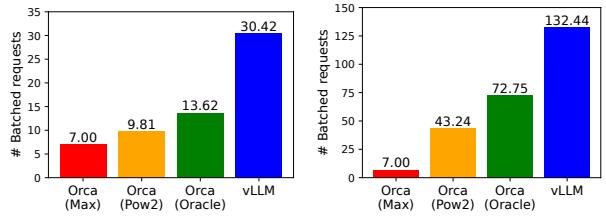


Figure 12. Single sequence generation with OPT models on the ShareGPT and Alpaca dataset



(a) ShareGPT **(b)** Alpaca

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 2×. 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.

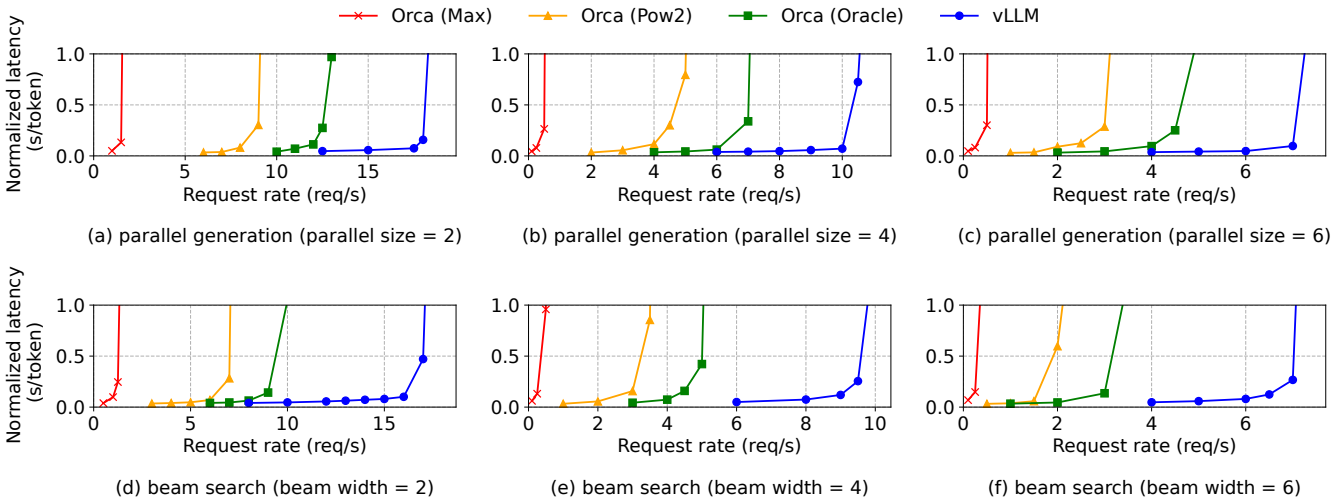


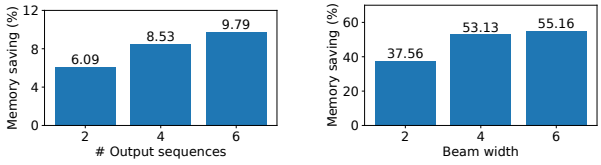
Figure 14. Parallel generation and beam search with OPT-13B on the Alpaca dataset.

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×–2.7× higher request rates compared to Orca (Oracle) and 2.7×–8× 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× more requests at the same time than Orca (Oracle) and 4.3× more requests than Orca (Max). Compared to FasterTransformer, vLLM can sustain up to 22× 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.



(a) Parallel sampling **(b)** Beam search

Figure 15. Average amount of memory saving from sharing KV blocks, when serving OPT-13B for the Alpaca trace.

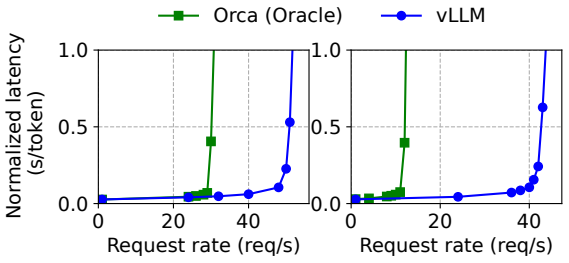
6.3 Parallel Sampling and Beam Search

We evaluate the effectiveness of memory sharing in PagedAttention 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× in basic sampling to 2.3× 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



(a) 1-shot prefix prompt (b) 5-shot prefix prompt

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.

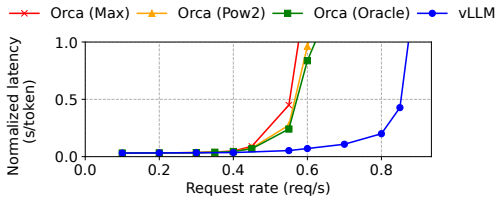


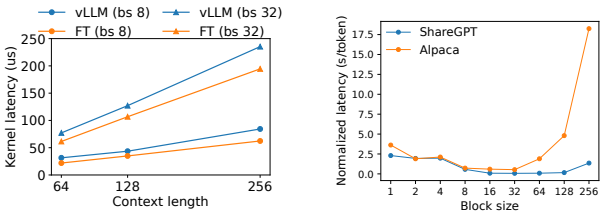
Figure 17. Performance on chatbot workload.

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× 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× 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× 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 handle the long prompts, as PagedAttention resolves the problem of memory fragmentation and reservation.



(a) Latency of attention kernels. **(b)** End-to-end latency with different block sizes.

Figure 18. Ablation experiments.

7 Ablation Studies

In this section, we study various aspects of vLLM and evaluate the design choices we make with ablation experiments.

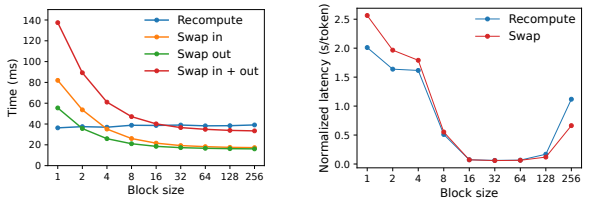
7.1 Kernel Microbenchmark

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.



(a) Microbenchmark **(b)** End-to-end performance

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.

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× 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] Ond rej 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, Guilio 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, Yinmin 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 Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-serving: Flexible, high-performance ml 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.

[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, Pierric 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, Shuohui 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, Zhuohan 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, Jiejing 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

HPC ASIA 2023, February 27-March 2, 2023, Singapore, Singapore

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 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[1].
- 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.

	V100 (SXM2)	V100S (PCIe)	A100 (SXM4/PCIe)	
SMs	80		108	
Clock [MHz]	1,380	1,597	1,410	
Device memory				
Size [GB]	32/16	32	40	80
Bandwidth [GB/s]	900	1,134	1,555	2,039
Shared memory				
Size [KB/SM]	~96		~164	
Bandwidth [GB/s]	14,131	16,353	19,491	
Performance				
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.

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

[1] 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 $\sim$ 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 $\mathbf{C} \leftarrow \mathbf{A} \cdot \mathbf{B}$ for $\mathbf{A} \in \mathbb{R}^{m \times k}, \mathbf{B} \in \mathbb{R}^{k \times n}$ is $2mnk$. On the other hand, the sum of the number of elements in $\mathbf{A}$ and $\mathbf{B}$ 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 $\mathbf{A}_b \cdot \mathbf{B}_b$ where the sizes of matrix $\mathbf{A}_b$ and $\mathbf{B}_b$ 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 $\mathbf{D} \leftarrow \mathbf{A} \cdot \mathbf{B} + \mathbf{C}$ on Tensor Cores using WMMA API for C++, first, we copy the input matrices $\mathbf{A}, \mathbf{B}$ and, $\mathbf{C}$ from memory to an array of registers

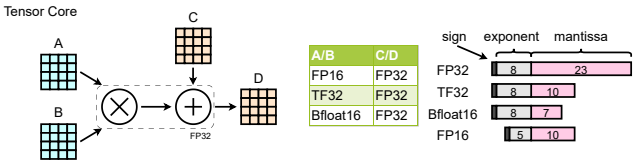


Figure 1. The input and output types of Tensor Cores on NVIDIA A100.

called "fragment". Then, we compute Matrix-Multiplication-and-Add (MMA) on the Tensor Cores and obtain the resulting $\mathbf{D}$ fragment. The 32 threads in a warp cooperate to perform MMA operations on Tensor Cores. Finally, we store the $\mathbf{D}$ 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.

```
__device__
void matmul(float* mem_c, half* mem_a, half* mem_b
    ) {
  using namespace nvcuda::wmma;
  fragment<matrix_a, 16, 16, 16, half, col_major>
    frag_a;
  fragment<matrix_b, 16, 16, 16, half, col_major>
    frag_b;
  fragment<accumulator, 16, 16, 16, float> frag_c;
  // Initialize an accumulator fragment
  fill_fragment(frag_c, 0.f);
  // Load matrices to fragments
  load_matrix_sync(frag_a, mem_a, ...);
  load_matrix_sync(frag_b, mem_b, ...);
  // Compute matrix-matrix multiplication
  // and accumulation on Tensor Cores
  mma_sync(frag_c, frag_a, frag_b, frag_c);
  // Store result to memory
  store_matrix_sync(mem_c, frag_c, ...);
}
```

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)

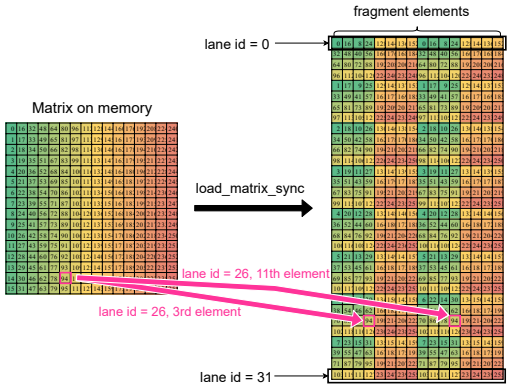


Figure 2. An example of memory-fragment mapping. The `lane id` is a thread number in a warp which is calculated by `(threadIdx.x & 0x1f)`.

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.

```
template <class Use, class Layout, class T>
__global__ void investigate_mapping() {
  __shared__ T smem[];
  // initialize smem
  for (i = 0; i < 16 * 16; i++) smem[i] = i;
  fragment<Use, 16, 16, 16, T, Layout> frag;
  load_matrix_sync(frag, smem, ...); // WMMA API
  for (i = 0; i < 32; i++) {
    if (threadIdx.x == i){
      for (j = 0; j < frag.num_elements; j++) {
        // Print the mapping
        printf("%d,", (int)frag.x[j]);}
      printf("\n");}
    __syncwarp();}}
```

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 ∼ 0.12 from Table 1. This value is similar to the ratio between the FP32 computing unit and device memory (0.06 ∼ 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 $\mathbf{A}_{\text{reg}}$ and $\mathbf{B}_{\text{reg}}$ for register blocking. The register blocking size is (n, n, n). We show the roofline model of computing $\mathbf{D}_{\text{reg}} \leftarrow \mathbf{A}_{\text{reg}} \cdot \mathbf{B}_{\text{reg}} + \mathbf{C}_{\text{reg}}$ in Figure 1. The Arithmetic Intensity (AI) is calculated as follows:

$$\text{AI} = \frac{2n^3}{(n^2+n^2)\text{sizeof(FP16)} + (n^2+n^2)\text{sizeof(FP32)}} = \frac{n}{5}. \tag{1}$$

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.

$$\text{nRegs} = ((\underbrace{n^2}_{\mathbf{A}_{\text{reg}}} + \underbrace{n^2}_{\mathbf{B}_{\text{reg}}}) \times \frac{1}{2} + \underbrace{n^2}_{\mathbf{C}_{\text{reg}}})/\text{warpSize} = \frac{1}{16}n^2. \tag{2}$$

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

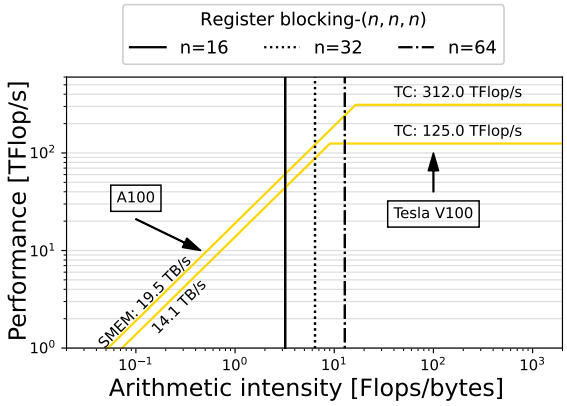


Figure 3. The arithmetic intensity of matrix-matrix multiplication for each size of register blocking blocking-(n, n, n).

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 $\begin{bmatrix} a_0 & a_1 & \cdots & a_{n-1} \end{bmatrix}$ using $n \times n$ matrix $\mathbf{U}$ as follows:

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}^\top \cdot \mathbf{U} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}^\top \cdot \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & 1 & \cdots & 1 \\ 0 & 0 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} = \begin{bmatrix} a_0 \\ \sum_{i=0}^{1} a_i \\ \vdots \\ \sum_{i=0}^{n-1} a_i \end{bmatrix}^\top.$$

The structural rule for the (i, j) element of the matrix $\mathbf{U}$ is as follows:

$$u_{i,j} = \begin{cases} 1 & i \leq j \\ 0 & \text{Otherwise} \end{cases} \tag{3}$$

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 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
      elements
4   [&](fid, i, j) {
5       if (i <= j) frag.x[fid] = 1;
6       else frag.x[fid] = 0;
7     });
```

Code 3. Generating the matrix U fragment from the structual 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 $\mathbf{A}$:

$$\mathbf{H} \cdot \mathbf{A} = \left(\mathbf{I}_m - 2\mathbf{v}^\top \mathbf{v}\right) \cdot \mathbf{A}, \tag{4}$$

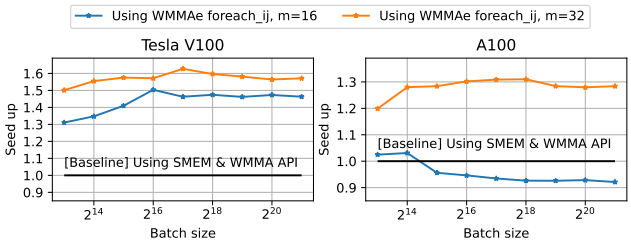


Figure 4. The performance evaluation of `foreach_ij` function using batched Householder benchmark, where we multiply a $m \times m$ Householder matrix $\mathbf{H}$ with an input matrix $\mathbf{A}$ using Tensor Cores.

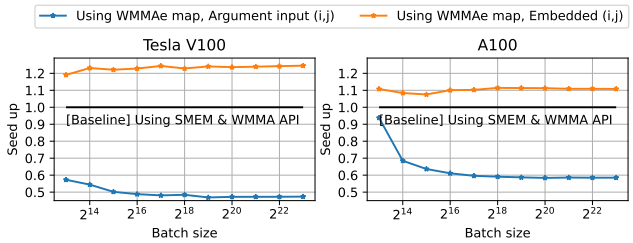


Figure 5. The performance evaluation of `map` function using batched Given's rotation benchmark. The "Argument input (i,j)" means that the parameter (i, j) for Given's rotation matrix is set through kernel function arguments, and "Embedded (i,j)" means that these parameters are set in compile-time.

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.

```
fragment<16, 16, 16> frag;
foreach_ij<decltype(frag)>(
  [&](fid, i, j) {
    auto elm = v[i]*v[j]*(-2);
    if (i==j) elm += 1
    frag.x[fid] = elm;
  });
```

Code 4. Generating a 16×16 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.

```
fragment<16, 16, 16> frag[2 * 2];// 32x32 matrix
foreach_ij<decltype(*frag)>(
  [&](fid, i, j) {
    for (unsigned bi = 0; bi < 2; bi++) {
      for (unsigned bi = 0; bi < 2; bi++) {
        auto elm = v[i+bi*16] * v[j+bj*16]*(-2);
        if (i==j) elm += 1
        frag[bi+bj*2].x[fid] = elm;
      }}});
```

Code 5. Generating an array of fragments for 32×32 Householder matrix using `foreach_ij`.

For the 32×32 matrix fragment, we used a 2×2 array of fragments holding matrices of size 16×16. 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 `(lid, 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.

```
fragment frag_a;
unsigned lid, fid;
// Calculate lid and fid from matrix position (i,
    j)
map<decltype(frag)>(lid, fid /*=2*/, i, j);
// Set 1
if ((threadIdx.x & 0x1f) == lid) {
  frag_a.x[fid] = 1;
}
```

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:

$$\mathbf{G}(i,j,\theta)\cdot\mathbf{A}, \tag{5}$$

where

$$\mathbf{G}(i,j,\theta)=\begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & \cdots & -s & \cdots & \\ & & \vdots & \ddots & \vdots & & \\ & & s & \cdots & c & \cdots & \\ & & \vdots & & \vdots & \ddots & \\ & & & & & & 1 \end{bmatrix} \begin{matrix} \\ \\ i\text{-th} \\ \\ j\text{-th} \\ \\ \\ \end{matrix},$$

$$\begin{matrix} & & i\text{-th} & & j\text{-th} & & \end{matrix}$$

$c=\cos\theta$ and $s=\sin\theta$. In this benchmark, Given's rotation operations for b FP16 input matrices $\mathbf{A}_k$ are performed by multiplying by $\mathbf{G}(i,j,\theta_k)$ 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 $\mathbf{G}$ 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 $\mathbf{G}$ 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 $\mathbf{C}_{F32}=\mathbf{A}_{F32}\mathbf{B}_{F32}$ as follows:

$$\begin{aligned} \mathbf{A}_{F16} &\leftarrow \text{toFP16}\left(\mathbf{A}_{F32}\right) && (6)\\ \Delta\mathbf{A}_{F16} &\leftarrow \text{toFP16}\left(\left(\mathbf{A}_{F32}-\text{toFP32}\left(\mathbf{A}_{F16}\right)\right)\times 2^{11}\right) \\ \mathbf{B}_{F16} &\leftarrow \text{toFP16}\left(\mathbf{B}_{F32}\right) && (7)\\ \Delta\mathbf{B}_{F16} &\leftarrow \text{toFP16}\left(\left(\mathbf{B}_{F32}-\text{toFP32}\left(\mathbf{B}_{F16}\right)\right)\times 2^{11}\right) \\ \mathbf{C}_{F32} &\leftarrow \mathbf{A}_{F16}\mathbf{B}_{F16}+\left(\Delta\mathbf{A}_{F16}\mathbf{B}_{F16}+\mathbf{A}_{F16}\Delta\mathbf{B}_{F16}\right)/2^{11}, \end{aligned}$$

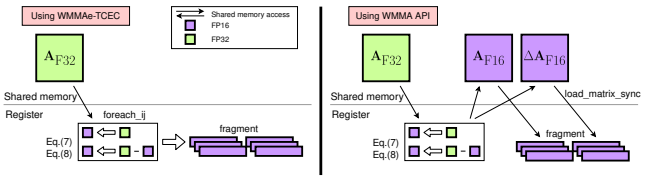


Figure 6. The comparison of data flow between using WMMA API and WMMAe. Here we load fragments for SGEMM emulation on Tensor Cores using error correction without additional shared memories $\mathbf{A}_{F16}$ and $\Delta\mathbf{A}_{F16}$, that are required when using WMMA API.

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 $\mathbf{A}_{F16}, \Delta\mathbf{A}_{F16}$ 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 $\mathbf{A}_{F16}$ and $\Delta\mathbf{A}_{F16}$ directly from the input matrix $\mathbf{A}_{F32}$ 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

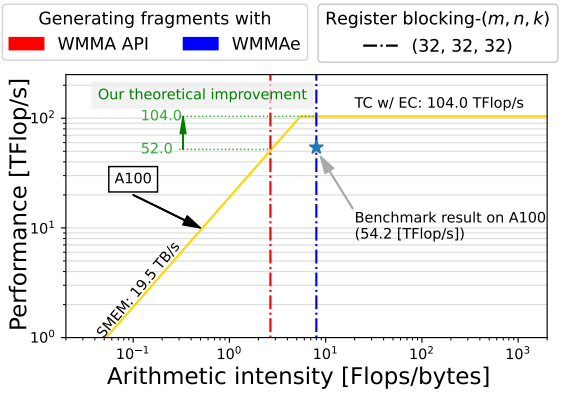


Figure 7. The arithmetic intensity of SGEMM emulation on Tensor Cores using error correction method. The peak performance is calculated by dividing the theoretical peak performance of FP16-TC in Table 1 by 3 since we need 3 times matrix-matrix multiplication in Eq. (8).

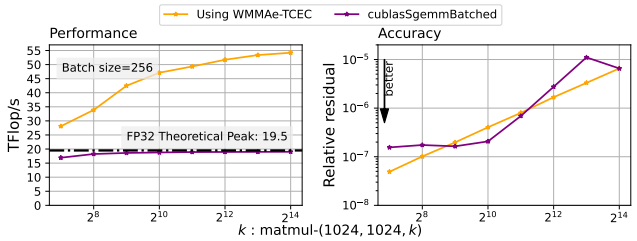


Figure 8. The throughput and accuracy evaluation of batched SGEMM using WMMAe-TCEC.

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

[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.

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 (RRNs) 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, RRNs 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[1].

[1] 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 $\mathbb{E}_N = \{\boldsymbol{x}_i\}_{i=1}^N$, where $\boldsymbol{x}_i \in \mathbb{R}^d$ 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.

$$\begin{aligned} \boldsymbol{q}_m &= f_q(\boldsymbol{x}_m, m) \\ \boldsymbol{k}_n &= f_k(\boldsymbol{x}_n, n) \\ \boldsymbol{v}_n &= f_v(\boldsymbol{x}_n, n), \end{aligned} \tag{1}$$

where $\boldsymbol{q}_m, \boldsymbol{k}_n$ and $\boldsymbol{v}_n$ 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.

$$
\begin{aligned}
a_{m,n} &= \frac{\exp(\frac{\boldsymbol{q}_m^\intercal \boldsymbol{k}_n}{\sqrt{d}})}{\sum_{j=1}^{N} \exp(\frac{\boldsymbol{q}_m^\intercal \boldsymbol{k}_j}{\sqrt{d}})} \\
\mathbf{o}_m &= \sum_{n=1}^{N} a_{m,n} \boldsymbol{v}_n
\end{aligned}
\tag{2}
$$

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

$$
f_{t:t\in\{q,k,v\}}(\boldsymbol{x}_i, i) := \boldsymbol{W}_{t:t\in\{q,k,v\}}(\boldsymbol{x}_i + \boldsymbol{p}_i), \tag{3}
$$

where $\boldsymbol{p}_i \in \mathbb{R}^d$ is a d-dimensional vector depending of the position of token $\boldsymbol{x}_i$. 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 $\boldsymbol{p}_i \in \{\boldsymbol{p}_t\}_{t=1}^{L}$, where L is the maximum sequence length. The authors of Vaswani et al. [2017] have proposed to generate $\boldsymbol{p}_i$ using the sinusoidal function.

$$
\begin{cases}
\boldsymbol{p}_{i,2t} &= \sin(k/10000^{2t/d}) \\
\boldsymbol{p}_{i,2t+1} &= \cos(k/10000^{2t/d})
\end{cases}
\tag{4}
$$

in which $\boldsymbol{p}_{i,2t}$ is the $2t^{th}$ element of the d-dimensional vector $\boldsymbol{p}_i$. 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:

$$
\begin{aligned}
f_q(\boldsymbol{x}_m) &:= \boldsymbol{W}_q \boldsymbol{x}_m \\
f_k(\boldsymbol{x}_n, n) &:= \boldsymbol{W}_k(\boldsymbol{x}_n + \tilde{\boldsymbol{p}}_r^k) \\
f_v(\boldsymbol{x}_n, n) &:= \boldsymbol{W}_v(\boldsymbol{x}_n + \tilde{\boldsymbol{p}}_r^v)
\end{aligned}
\tag{5}
$$

where $\tilde{\boldsymbol{p}}_r^k, \tilde{\boldsymbol{p}}_r^v \in \mathbb{R}^d$ are trainable relative position embeddings. Note that $r = \text{clip}(m-n, r_{\text{min}}, r_{\text{max}})$ 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 $\boldsymbol{q}_m^\intercal \boldsymbol{k}_n$ of Equation (2) as

$$
\boldsymbol{q}_m^\intercal \boldsymbol{k}_n = \boldsymbol{x}_m^\intercal \boldsymbol{W}_q^\intercal \boldsymbol{W}_k \boldsymbol{x}_n + \boldsymbol{x}_m^\intercal \boldsymbol{W}_q^\intercal \boldsymbol{W}_k \boldsymbol{p}_n + \boldsymbol{p}_m^\intercal \boldsymbol{W}_q^\intercal \boldsymbol{W}_k \boldsymbol{x}_n + \boldsymbol{p}_m^\intercal \boldsymbol{W}_q^\intercal \boldsymbol{W}_k \boldsymbol{p}_n, \tag{6}
$$

the key idea is to replace the absolute position embedding $\boldsymbol{p}_n$ with its sinusoid-encoded relative counterpart $\tilde{\boldsymbol{p}}_{m-n}$, while the absolute position $\boldsymbol{p}_m$ in the third and fourth term with two trainable vectors $\mathbf{u}$ and $\mathbf{v}$ independent of the query positions. Further, $\boldsymbol{W}_k$ is distinguished for the content-based and location-based key vectors $\boldsymbol{x}_n$ and $\boldsymbol{p}_n$, denoted as $\boldsymbol{W}_k$ and $\widetilde{\boldsymbol{W}}_k$, resulting in:

$$
\boldsymbol{q}_m^\intercal \boldsymbol{k}_n = \boldsymbol{x}_m^\intercal \boldsymbol{W}_q^\intercal \boldsymbol{W}_k \boldsymbol{x}_n + \boldsymbol{x}_m^\intercal \boldsymbol{W}_q^\intercal \widetilde{\boldsymbol{W}}_k \tilde{\boldsymbol{p}}_{m-n} + \mathbf{u}^\intercal \boldsymbol{W}_q^\intercal \boldsymbol{W}_k \boldsymbol{x}_n + \mathbf{v}^\intercal \boldsymbol{W}_q^\intercal \widetilde{\boldsymbol{W}}_k \tilde{\boldsymbol{p}}_{m-n} \tag{7}
$$

It is noteworthy that the position information in the value term is removed by setting $f_v(\boldsymbol{x}_j) := \boldsymbol{W}_v \boldsymbol{x}_j$. 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:

$$
\boldsymbol{q}_m^\intercal \boldsymbol{k}_n = \boldsymbol{x}_m^\intercal \boldsymbol{W}_q^\intercal \boldsymbol{W}_k \boldsymbol{x}_n + b_{i,j} \tag{8}
$$

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.

$$
\boldsymbol{q}_m^\intercal \boldsymbol{k}_n = \boldsymbol{x}_m^\intercal \boldsymbol{W}_q^\intercal \boldsymbol{W}_k \boldsymbol{x}_n + \boldsymbol{p}_m^\intercal \mathbf{U}_q^\intercal \mathbf{U}_k \boldsymbol{p}_n + b_{i,j} \tag{9}
$$

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 $\boldsymbol{p}_m$ and $\boldsymbol{p}_n$ were simply replaced with the relative position embeddings $\tilde{\boldsymbol{p}}_{m-n}$:

$$\boldsymbol{q}_m^\intercal \boldsymbol{k}_n = \boldsymbol{x}_m^\intercal \boldsymbol{W}_q^\intercal \boldsymbol{W}_k \boldsymbol{x}_n + \boldsymbol{x}_m^\intercal \boldsymbol{W}_q^\intercal \boldsymbol{W}_k \tilde{\boldsymbol{p}}_{m-n} + \tilde{\boldsymbol{p}}_{m-n}^\intercal \boldsymbol{W}_q^\intercal \boldsymbol{W}_k \boldsymbol{x}_n \tag{10}$$

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), $\boldsymbol{q}_m^\intercal \boldsymbol{k}_n$ typically enables knowledge conveyance between tokens at different positions. In order to incorporate relative position information, we require the inner product of query $\boldsymbol{q}_m$ and key $\boldsymbol{k}_n$ to be formulated by a function g, which takes only the word embeddings $\boldsymbol{x}_m$, $\boldsymbol{x}_n$, 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:

$$\langle f_q(\boldsymbol{x}_m, m), f_k(\boldsymbol{x}_n, n) \rangle = g(\boldsymbol{x}_m, \boldsymbol{x}_n, m - n). \tag{11}$$

The ultimate goal is to find an equivalent encoding mechanism to solve the functions $f_q(\boldsymbol{x}_m, m)$ and $f_k(\boldsymbol{x}_n, n)$ 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:

$$\begin{aligned} f_q(\boldsymbol{x}_m, m) &= (\boldsymbol{W}_q \boldsymbol{x}_m) e^{im\theta} \\ f_k(\boldsymbol{x}_n, n) &= (\boldsymbol{W}_k \boldsymbol{x}_n) e^{in\theta} \\ g(\boldsymbol{x}_m, \boldsymbol{x}_n, m - n) &= \mathrm{Re}[(\boldsymbol{W}_q \boldsymbol{x}_m)(\boldsymbol{W}_k \boldsymbol{x}_n)^* e^{i(m-n)\theta}] \end{aligned} \tag{12}$$

where $\mathrm{Re}[\cdot]$ is the real part of a complex number and $(\boldsymbol{W}_k \boldsymbol{x}_n)^*$ represents the conjugate complex number of $(\boldsymbol{W}_k \boldsymbol{x}_n)$. $\theta \in \mathbb{R}$ is a preset non-zero constant. We can further write $f_{\{q,k\}}$ in a multiplication matrix:

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} W_{\{q,k\}}^{(11)} & W_{\{q,k\}}^{(12)} \\ W_{\{q,k\}}^{(21)} & W_{\{q,k\}}^{(22)} \end{pmatrix} \begin{pmatrix} x_m^{(1)} \\ x_m^{(2)} \end{pmatrix} \tag{13}$$

where $(x_m^{(1)}, x_m^{(2)})$ is $\boldsymbol{x}_m$ 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 $\boldsymbol{x}_i \in \mathbb{R}^d$ 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:

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \boldsymbol{R}^d_{\Theta,m} \boldsymbol{W}_{\{q,k\}} \boldsymbol{x}_m \tag{14}$$

where

$$\boldsymbol{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix} \tag{15}$$

is the rotary matrix with pre-defined parameters $\Theta = \{\theta_i = 10000^{-2(i-1)/d}, i \in [1, 2, ..., d/2]\}$. A graphic illustration of RoPE is shown in Figure **(1)**. Applying our RoPE to self-attention in Equation (2), we obtain:

$$\boldsymbol{q}_m^\intercal \boldsymbol{k}_n = (\boldsymbol{R}^d_{\Theta,m} \boldsymbol{W}_q \boldsymbol{x}_m)^\intercal (\boldsymbol{R}^d_{\Theta,n} \boldsymbol{W}_k \boldsymbol{x}_n) = \boldsymbol{x}^\intercal \boldsymbol{W}_q R^d_{\Theta,n-m} \boldsymbol{W}_k \boldsymbol{x}_n \tag{16}$$

where $\boldsymbol{R}^d_{\Theta,n-m} = (\boldsymbol{R}^d_{\Theta,m})^\intercal \boldsymbol{R}^d_{\Theta,n}$. Note that $\boldsymbol{R}^d_\Theta$ is an orthogonal matrix, which ensures stability during the process of encoding position information. In addition, due to the sparsity of R^d_Θ, 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.

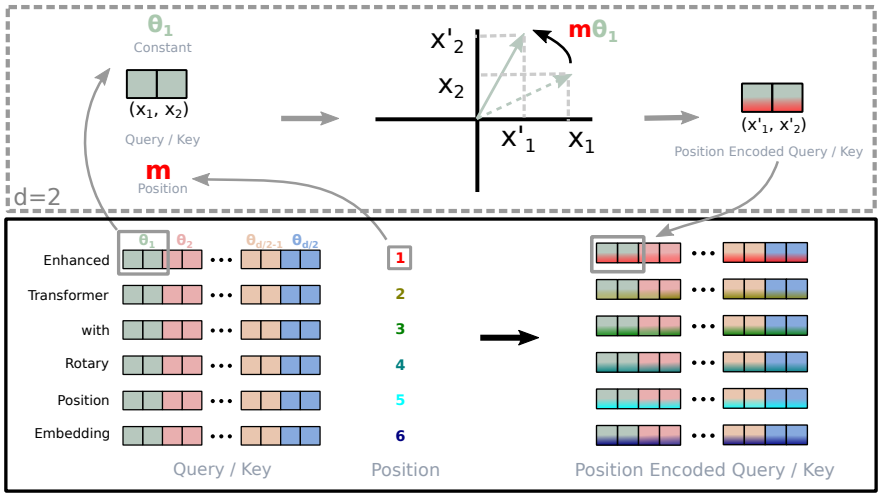


Figure 1: Implementation of Rotary Position Embedding(RoPE).

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.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})_m = \frac{\sum_{n=1}^N \text{sim}(\boldsymbol{q}_m, \boldsymbol{k}_n) \boldsymbol{v}_n}{\sum_{n=1}^N \text{sim}(\boldsymbol{q}_m, \boldsymbol{k}_n)}. \tag{17}$$

The original self-attention chooses $\mathrm{sim}(\boldsymbol{q}_m, \boldsymbol{k}_n) = \exp(\boldsymbol{q}_m^\intercal \boldsymbol{k}_n / \sqrt{d})$. 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 $\mathbb{O}(N^2)$. Follow Katharopoulos et al. [2020], the linear attentions reformulate Equation (17) as

$$\mathrm{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})_m = \frac{\sum_{n=1}^{N} \phi(\boldsymbol{q}_m)^\intercal \varphi(\boldsymbol{k}_n) \boldsymbol{v}_n}{\sum_{n=1}^{N} \phi(\boldsymbol{q}_m)^\intercal \varphi(\boldsymbol{k}_n)}, \tag{18}$$

where $\phi(\cdot), \varphi(\cdot)$ are usually non-negative functions. The authors of Katharopoulos et al. [2020] have proposed $\phi(x) = \varphi(x) = \mathrm{elu}(x) + 1$ 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 $\phi(\boldsymbol{q}_i) = \mathrm{softmax}(\boldsymbol{q}_i)$ and $\phi(\boldsymbol{k}_j) = \exp(\boldsymbol{k}_j)$. 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.

$$\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})_m = \frac{\sum_{n=1}^{N} \left(\boldsymbol{R}^d_{\Theta,m} \phi(\boldsymbol{q}_m)\right)^\intercal \left(\boldsymbol{R}^d_{\Theta,n} \varphi(\boldsymbol{k}_n)\right) \boldsymbol{v}_n}{\sum_{n=1}^{N} \phi(\boldsymbol{q}_m)^\intercal \varphi(\boldsymbol{k}_n)}. \tag{19}$$

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 $\boldsymbol{v}_i$ 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 $\boldsymbol{x}_q$, $\boldsymbol{x}_k$ corresponds to query and key and their position m and n, respectively. According to eq. (1), their position-encoded counterparts are:

$$\begin{aligned} \boldsymbol{q}_m &= f_q(\boldsymbol{x}_q, m), \\ \boldsymbol{k}_n &= f_k(\boldsymbol{x}_k, n), \end{aligned} \tag{20}$$

where the subscripts of $\boldsymbol{q}_m$ and $\boldsymbol{k}_n$ indicate the encoded positions information. Assume that there exists a function g that defines the inner product between vectors produced by $f_{\{q,k\}}$:

$$\boldsymbol{q}_m^\intercal \boldsymbol{k}_n = \langle f_q(\boldsymbol{x}_m, m), f_k(\boldsymbol{x}_n, n) \rangle = g(\boldsymbol{x}_m, \boldsymbol{x}_n, n - m), \tag{21}$$

we further require below initial condition to be satisfied:

$$\begin{aligned} \boldsymbol{q} &= f_q(\boldsymbol{x}_q, 0), \\ \boldsymbol{k} &= f_k(\boldsymbol{x}_k, 0), \end{aligned} \tag{22}$$

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:

$$\begin{aligned} f_q(\boldsymbol{x}_q, m) &= R_q(\boldsymbol{x}_q, m) e^{i\Theta_q(\boldsymbol{x}_q, m)}, \\ f_k(\boldsymbol{x}_k, n) &= R_k(\boldsymbol{x}_k, n) e^{i\Theta_k(\boldsymbol{x}_k, n)}, \\ g(\boldsymbol{x}_q, \boldsymbol{x}_k, n - m) &= R_g(\boldsymbol{x}_q, \boldsymbol{x}_k, n - m) e^{i\Theta_g(\boldsymbol{x}_q, \boldsymbol{x}_k, n - m)}, \end{aligned} \tag{23}$$

where R_f, R_g and Θ_f, Θ_g are the radical and angular components for $f_{\{q,k\}}$ and g, respectively. Plug them into Equation (21), we get the relation:

$$\begin{aligned} R_q(\boldsymbol{x}_q, m) R_k(\boldsymbol{x}_k, n) &= R_g(\boldsymbol{x}_q, \boldsymbol{x}_k, n - m), \\ \Theta_k(\boldsymbol{x}_k, n) - \Theta_q(\boldsymbol{x}_q, m) &= \Theta_g(\boldsymbol{x}_q, \boldsymbol{x}_k, n - m), \end{aligned} \tag{24}$$

with the corresponding initial condition as:

$$\begin{aligned} \boldsymbol{q} &= \|\boldsymbol{q}\| e^{i\theta_q} = R_q(\boldsymbol{x}_q, 0) e^{i\Theta_q(\boldsymbol{x}_q, 0)}, \\ \boldsymbol{k} &= \|\boldsymbol{k}\| e^{i\theta_k} = R_k(\boldsymbol{x}_k, 0) e^{i\Theta_k(\boldsymbol{x}_k, 0)}, \end{aligned} \tag{25}$$

where $\|\boldsymbol{q}\|$, $\|\boldsymbol{k}\|$ and θ_q, θ_k are the radial and angular part of $\boldsymbol{q}$ and $\boldsymbol{k}$ on the 2D plane.

Next, we set $m = n$ in Equation (24) and take into account initial conditions in Equation (25):

$$R_q(\boldsymbol{x}_q, m)R_k(\boldsymbol{x}_k, m) = R_q(\boldsymbol{x}_q, \boldsymbol{x}_k, 0) = R_k(\boldsymbol{x}_q, 0)R_k(\boldsymbol{x}_k, 0) = \|\boldsymbol{q}\|\|\boldsymbol{k}\|, \tag{26a}$$

$$\Theta_k(\boldsymbol{x}_k, m) - \Theta_q(\boldsymbol{x}_q, m) = \Theta_g(\boldsymbol{x}_q, \boldsymbol{x}_k, 0) = \|\Theta_k(\boldsymbol{x}_k, 0) - \Theta_q(\boldsymbol{x}_q, 0)\| = \|\theta_k - \theta_q\|. \tag{26b}$$

On one hand, from, a straightforward solution of R_f could be formed from Equation (26a) :

$$\begin{aligned} R_q(\boldsymbol{x}_q, m) &= R_q(\boldsymbol{x}_q, 0) = \|\boldsymbol{q}\| \\ R_k(\boldsymbol{x}_k, n) &= R_k(\boldsymbol{x}_k, 0) = \|\boldsymbol{k}\| \\ R_g(\boldsymbol{x}_q, \boldsymbol{x}_k, n-m) &= R_g(\boldsymbol{x}_q, \boldsymbol{x}_k, 0) = \|\boldsymbol{q}\|\|\boldsymbol{k}\| \end{aligned} \tag{27}$$

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), $\Theta_q(\boldsymbol{x}_q, m) - \theta_q = \Theta_k(\boldsymbol{x}_k, m) - \theta_k$ indicates that the angular functions does not dependent on query and key, we set them to $\Theta_f := \Theta_q = \Theta_k$ and term $\Theta_f(\boldsymbol{x}_{\{q,k\}}, m) - \theta_{\{q,k\}}$ is a function of position m and is independent of word embedding $\boldsymbol{x}_{\{q,k\}}$, we denote it as $\phi(m)$, yielding:

$$\Theta_f(\boldsymbol{x}_{\{q,k\}}, m) = \phi(m) + \theta_{\{q,k\}}, \tag{28}$$

Further, by plugging $n = m + 1$ to Equation (24) and consider the above equation, we can get:

$$\phi(m+1) - \phi(m) = \Theta_g(\boldsymbol{x}_q, \boldsymbol{x}_k, 1) + \theta_q - \theta_k, \tag{29}$$

Since RHS is a constant irrelevant to m, $\phi(m)$ with continuous integer inputs produce an arithmetic progression:

$$\phi(m) = m\theta + \gamma, \tag{30}$$

where $\theta, \gamma \in \mathbb{R}$ are constants and θ is non-zero. To summarize our solutions from Equations (27) to (30):

$$\begin{aligned} f_q(\boldsymbol{x}_q, m) &= \|\boldsymbol{q}\|e^{i\theta_q + m\theta + \gamma} = \boldsymbol{q}e^{i(m\theta+\gamma)}, \\ f_k(\boldsymbol{x}_k, n) &= \|\boldsymbol{k}\|e^{i\theta_k + n\theta + \gamma} = \boldsymbol{k}e^{i(n\theta+\gamma)}. \end{aligned} \tag{31}$$

Note that we do not apply any constrains to f_q and f_k of Equation (22), thus $f_q(\boldsymbol{x}_m, 0)$ and $f_k(\boldsymbol{x}_n, 0)$ are left to choose freely. To make our results comparable to Equation (3), we define:

$$\begin{aligned} \boldsymbol{q} &= f_q(\boldsymbol{x}_m, 0) = \boldsymbol{W}_q\boldsymbol{x}_n, \\ \boldsymbol{k} &= f_k(\boldsymbol{x}_n, 0) = \boldsymbol{W}_k\boldsymbol{x}_n. \end{aligned} \tag{32}$$

Then, we simply set $\gamma = 0$ in Equation (31) of the final solution:

$$\begin{aligned} f_q(\boldsymbol{x}_m, m) &= (\boldsymbol{W}_q\boldsymbol{x}_m)e^{im\theta}, \\ f_k(\boldsymbol{x}_n, n) &= (\boldsymbol{W}_k\boldsymbol{x}_n)e^{in\theta}. \end{aligned} \tag{33}$$

3.4.2 Computational efficient realization of rotary matrix multiplication

Taking the advantage of the sparsity of $\boldsymbol{R}^d_{\Theta,m}$ in Equation (15), a more computational efficient realization of a multiplication of R^d_Θ and $\boldsymbol{x} \in \mathbb{R}^d$ is:

$$\boldsymbol{R}^d_{\Theta,m}\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix} \tag{34}$$

3.4.3 Long-term decay of RoPE

We can group entries of vectors $\boldsymbol{q} = \boldsymbol{W}_q\boldsymbol{x}_m$ and $\boldsymbol{k} = \boldsymbol{W}_k\boldsymbol{x}_n$ in pairs, and the inner product of RoPE in Equation (16) can be written as a complex number multiplication.

$$(\boldsymbol{R}^d_{\Theta,m}\boldsymbol{W}_q\boldsymbol{x}_m)^\intercal(\boldsymbol{R}^d_{\Theta,n}\boldsymbol{W}_k\boldsymbol{x}_n) = \mathrm{Re}\left[\sum_{i=0}^{d/2-1} \boldsymbol{q}_{[2i:2i+1]}\boldsymbol{k}^*_{[2i:2i+1]}e^{i(m-n)\theta_i}\right] \tag{35}$$

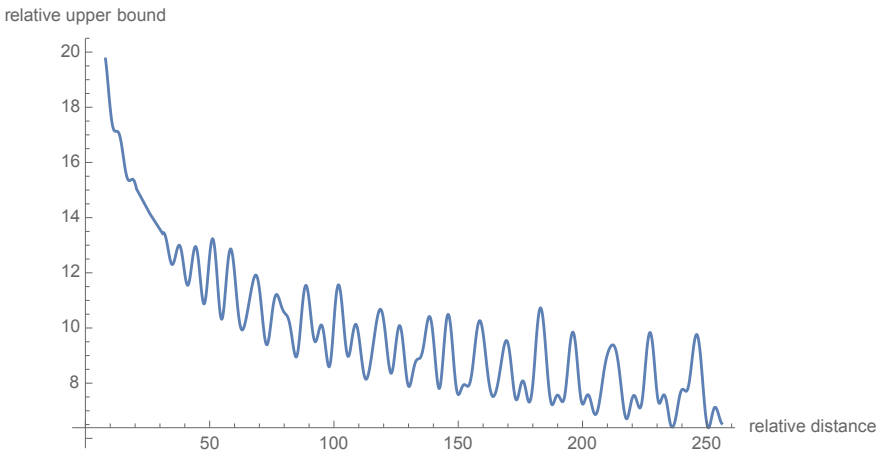


Figure 2: Long-term decay of RoPE.

where $\boldsymbol{q}_{[2i:2i+1]}$ represents the $2i^{th}$ to $(2i+1)^{th}$ entries of $\boldsymbol{q}$. Denote $h_i = \boldsymbol{q}_{[2i:2i+1]}\boldsymbol{k}^*_{[2i:2i+1]}$ 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

$$\sum_{i=0}^{d/2-1} \boldsymbol{q}_{[2i:2i+1]}\boldsymbol{k}^*_{[2i:2i+1]}e^{i(m-n)\theta_i} = \sum_{i=0}^{d/2-1} h_i(S_{i+1} - S_i) = -\sum_{i=0}^{d/2-1} S_{i+1}(h_{i+1} - h_i). \tag{36}$$

Thus,

$$\begin{aligned}\left| \sum_{i=0}^{d/2-1} \boldsymbol{q}_{[2i:2i+1]}\boldsymbol{k}^*_{[2i:2i+1]}e^{i(m-n)\theta_i} \right| &= \left| \sum_{i=0}^{d/2-1} S_{i+1}(h_{i+1} - h_i) \right| \\ &\leq \sum_{i=0}^{d/2-1} |S_{i+1}||(h_{i+1} - h_i)| \\ &\leq \left(\max_i |h_{i+1} - h_i|\right) \sum_{i=0}^{d/2-1} |S_{i+1}|\end{aligned} \tag{37}$$

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 benchmarksSingh 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 severs 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	**27.5**

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 BLEUPapineni et al. [2002] 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.

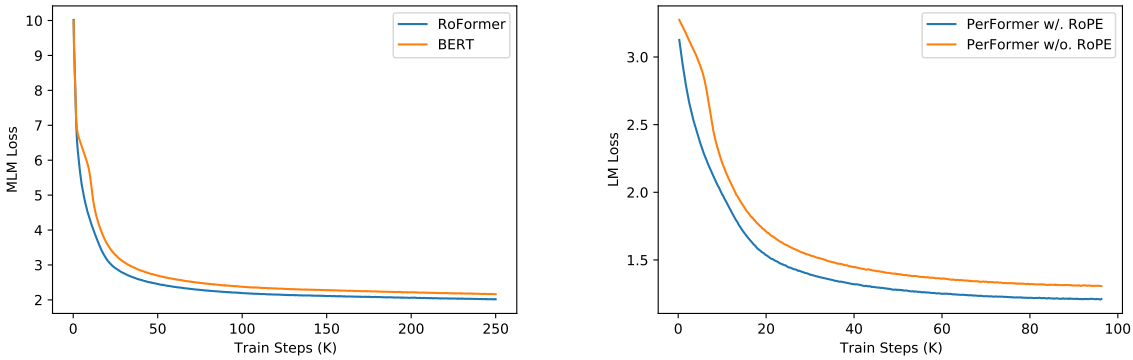


Figure 3: Evaluation of RoPE in language modeling pre-training. **Left**: training loss for BERT and RoFormer. **Right**: training loss for PerFormer with and without RoPE.

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	**89.5**	90.7	88.0	**87.0**	**86.4**	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[2]. 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.

[2]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%
RoFormer-512	64.13%	68.29%
RoFormer-1024	**66.07**%	**69.79**%

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 absolution 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, L ukasz 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/3f5ee243547dee91fbd053c1c4a845aa-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=ByxRM0Ntvr`.

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. Dinculescu, 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-WTVtwr.

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 Likhosherstov, 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, Pierric 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 [*1] **Ji Lin** [*1] **Mickael Seznec** [2] **Hao Wu** [2] **Julien Demouth** [2] **Song Han** [1]

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 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.

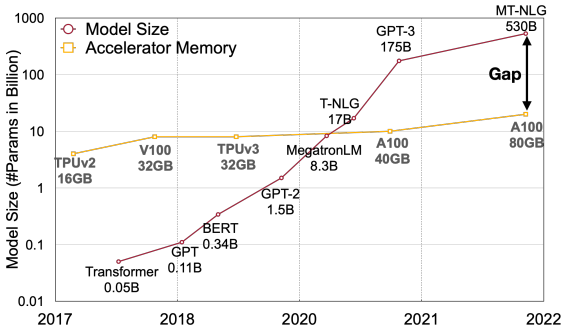


Figure 1: The model size of large language models is developing at a faster pace than the GPU memory in recent years, leading to a big gap between the supply and demand for memory. Quantization and model compression techniques SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language ModelsSmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Modelscan help bridge the gap.

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

[*]Equal contribution [1]Massachusetts Institute of Technology [2]NVIDIA. Correspondence to: Guangxuan Xiao <xgx@mit.edu>, Ji Lin <jilin@mit.edu>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

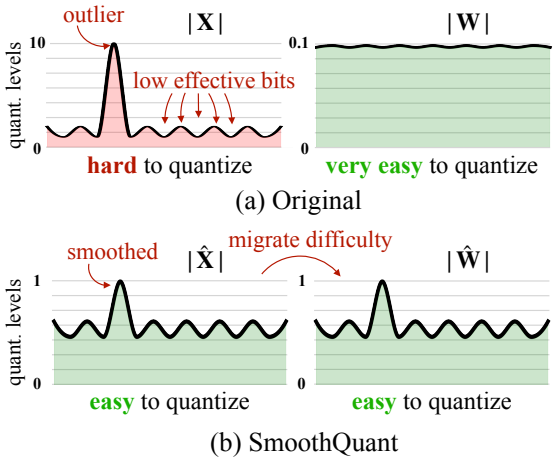


Figure 2: SmoothQuant's intuition: the activation $\mathbf{X}$ is hard to quantize because outliers stretch the quantization range, leaving few effective bits for most values. We migrate the scale variance from activations to weights $\mathbf{W}$ during offline to reduce the quantization difficulty of 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× speed up and 1.96× 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× 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

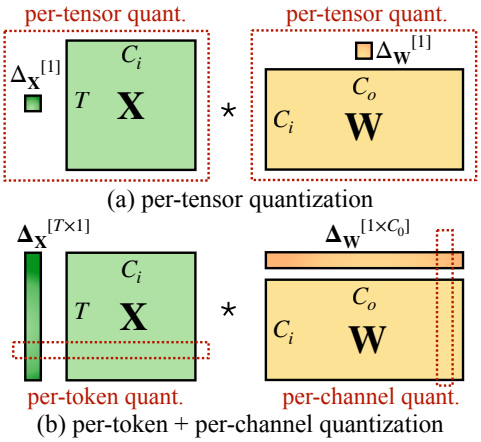
Quantization maps a high-precision value into discrete levels. We study integer uniform quantization (Jacob et al., 2018) (specifically INT8) for better hardware support and efficiency. The quantization process can be expressed as:

$$\bar{\mathbf{X}}^{\text{INT8}} = \lceil \frac{\mathbf{X}^{\text{FP16}}}{\Delta} \rfloor, \quad \Delta = \frac{\max(|\mathbf{X}|)}{2^{N-1}-1}, \tag{1}$$

where $\mathbf{X}$ is the floating-point tensor, $\bar{\mathbf{X}}$ is the quantized counterpart, Δ is the quantization step size, $\lceil \cdot \rfloor$ 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 Δ so that it preserves the outliers in activation, which are found to be important for accuracy (Dettmers et al., 2022). We can calculate Δ offline with the activations of some calibration samples, what we call **static quantization**. We can also use the runtime statistics of activations to get Δ, 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.).



(a) per-tensor quantization

(b) per-token + per-channel quantization

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 (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 echos the findings of Bondarenko et al..

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%

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:

$$\mathbf{Y} = \text{diag}(\mathbf{\Delta}_{\mathbf{X}}^{\text{FP16}}) \cdot (\bar{\mathbf{X}}^{\text{INT8}} \cdot \bar{\mathbf{W}}^{\text{INT8}}) \cdot \text{diag}(\mathbf{\Delta}_{\mathbf{W}}^{\text{FP16}}) \quad (2)$$

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

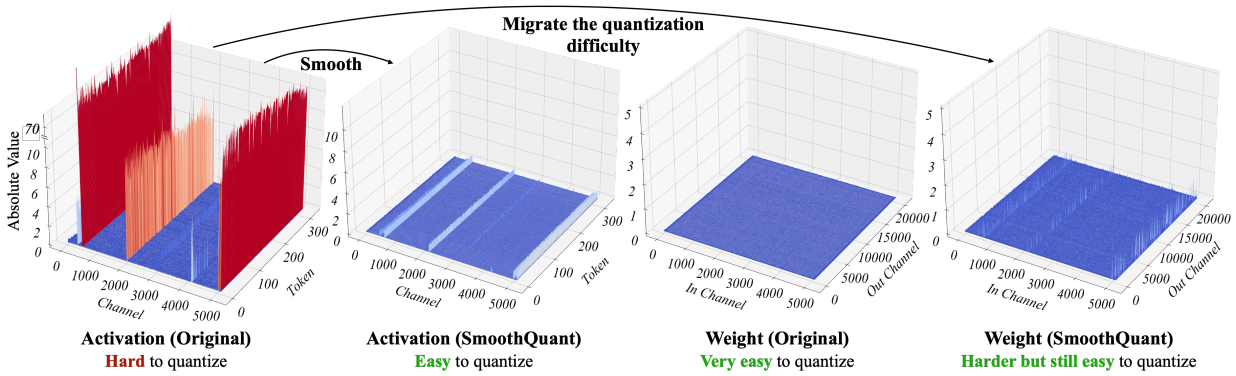


Figure 4: Magnitude of the input activations and weights of a linear layer in OPT-13B before and after SmoothQuant. Observations: (1) there are a few channels in the original activation map whose magnitudes are very large (greater than 70); (2) the variance in one activation channel is small; (3) the 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 doe 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 $\mathbf{s}$ 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 $\mathbf{s}_j = \max(|\mathbf{X}_j|), j = 1, 2, ..., C_i$, 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 $\mathbf{s}_j = 1/\max(|\mathbf{W}_j|)$. 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 α, to control how much difficulty we want to migrate from activation to weights, using the following equation:

$$\mathbf{s}_j = \max(|\mathbf{X}_j|)^{\alpha} / \max(|\mathbf{W}_j|)^{1-\alpha} \tag{4}$$

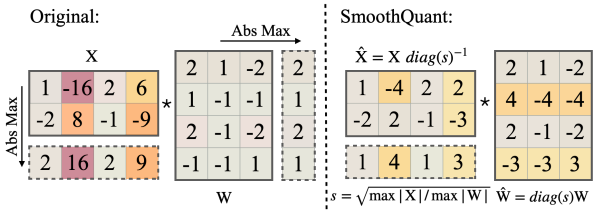


Figure 5: Main idea of SmoothQuant when α is 0.5. The smoothing factor s is obtained on calibration samples and the entire transformation is performed offline. At runtime, the activations are smooth without scaling.

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 ~30% outliers, which are more difficult for activation quantization), we can choose a larger α 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,

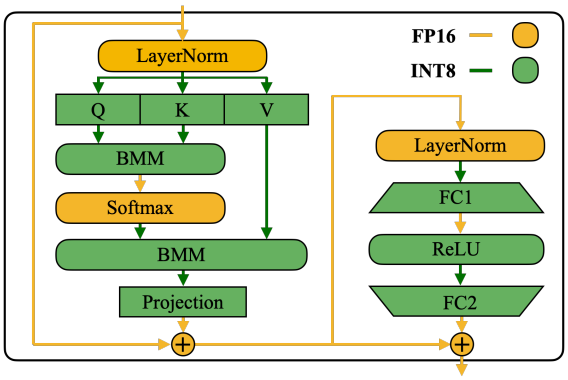


Figure 6: SmoothQuant's precision mapping for a Transformer block. All compute-intensive operators like linear layers and batched matmul (`BMM`s) use INT8 arithmetic.

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 LayerNorm. 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 α 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.

OPT-175B	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	**71.2%**	68.3%	**73.7%**
SmoothQuant-O2	71.1%	**68.4%**	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.). Nonetheless, 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.

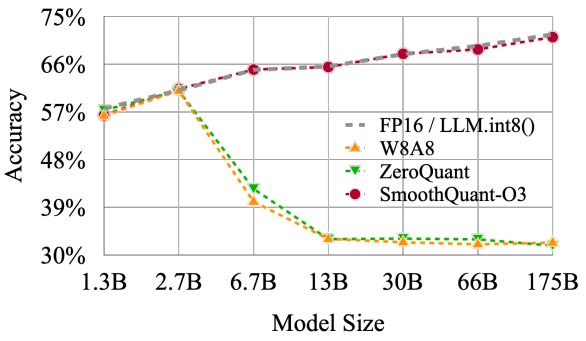


Figure 7: SmoothQuant-O3 (the most efficient setting, defined in Table 2) preserves the accuracy of OPT models across different scales when quantized to INT8. LLM.int8() requires mixed precision and suffers from slowing down.

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 ↑	WikiText ↓
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	**69.77%**	**14.37**

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 α=0.8 for SmoothQuant.

Wiki PPL↓	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 languange 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

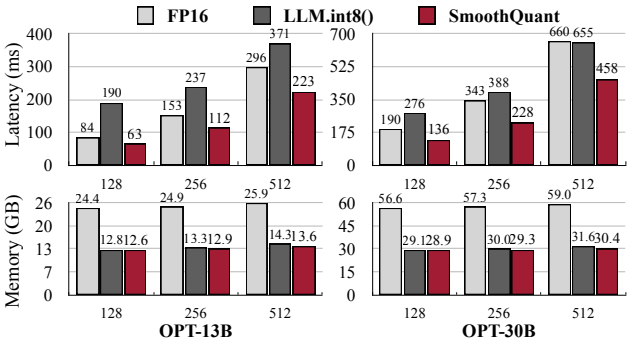


Figure 8: The PyTorch implementation of SmoothQuant-O3 achieves up to **1.51**× speedup and **1.96**× memory saving for OPT models on a single NVIDIA A100-80GB GPU, while `LLM.int8()` slows down the inference in most cases.

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 `GEMM`s.

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× when using a single GPU. This is challenging since FasterTransformer is already more than 3× 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 2×, as shown on Figure 9 (bottom).

Decoding-stage. In Table 7, we show SmoothQuant can significantly accelerate the autoregressive decoding stage

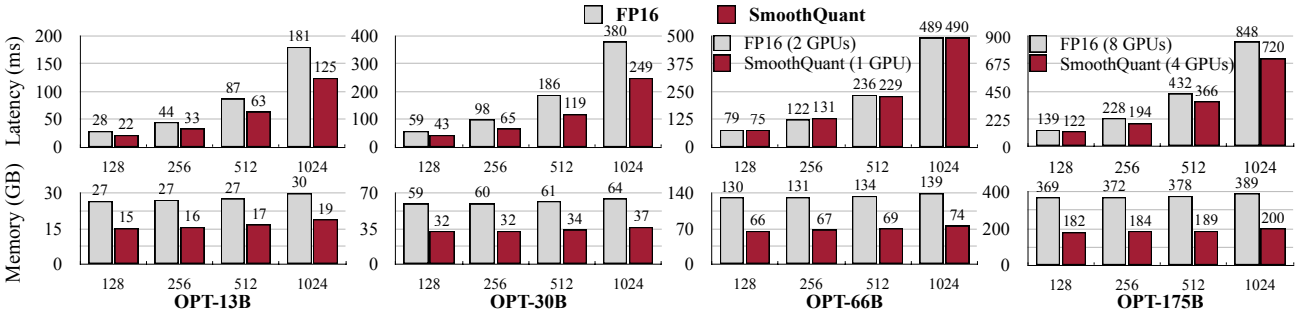


Figure 9: Inference latency (top) and memory usage (bottom) of the FasterTransformer implementation on NVIDIA A100-80GB GPUs. For smaller models, the latency can be significantly reduced with SmoothQuant-O3 by up to 1.56x compared to FP16. For the bigger models (OPT-66B and 175B), we can achieve similar 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	Saving (↑)
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×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 α (see Equation 4) to balance the quantization difficulty of weights and activations. We ablate the effect of different α's on OPT-175B with LAMBADA in Figure 10. When α is too small (<0.4), the activations are hard to quantize; when α is too large (>0.6), the weights will be hard to quantize. Only when we choose α 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	
Sequence Length	256	512	256	512
FP16	152.6	296.3	343.0	659.9
`LLM.int8()`	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

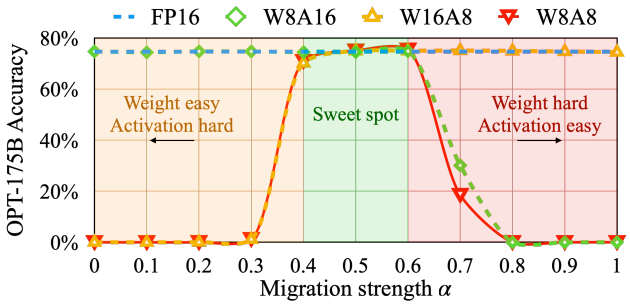


Figure 10: A suitable migration strength α (sweet spot) makes both activations and weights easy to quantize. If the α is too large, weights will be hard to quantize; if too small, activations will be hard to quantize.

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 works (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 equalization (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). ZeroQuant (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 LayerNorm 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 `GEMM`s 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× 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., Hoefler, 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., Mihaylov, 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 kenerl [¶] 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.

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.

[¶] https://github.com/IST-DASLab/gptq

SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models

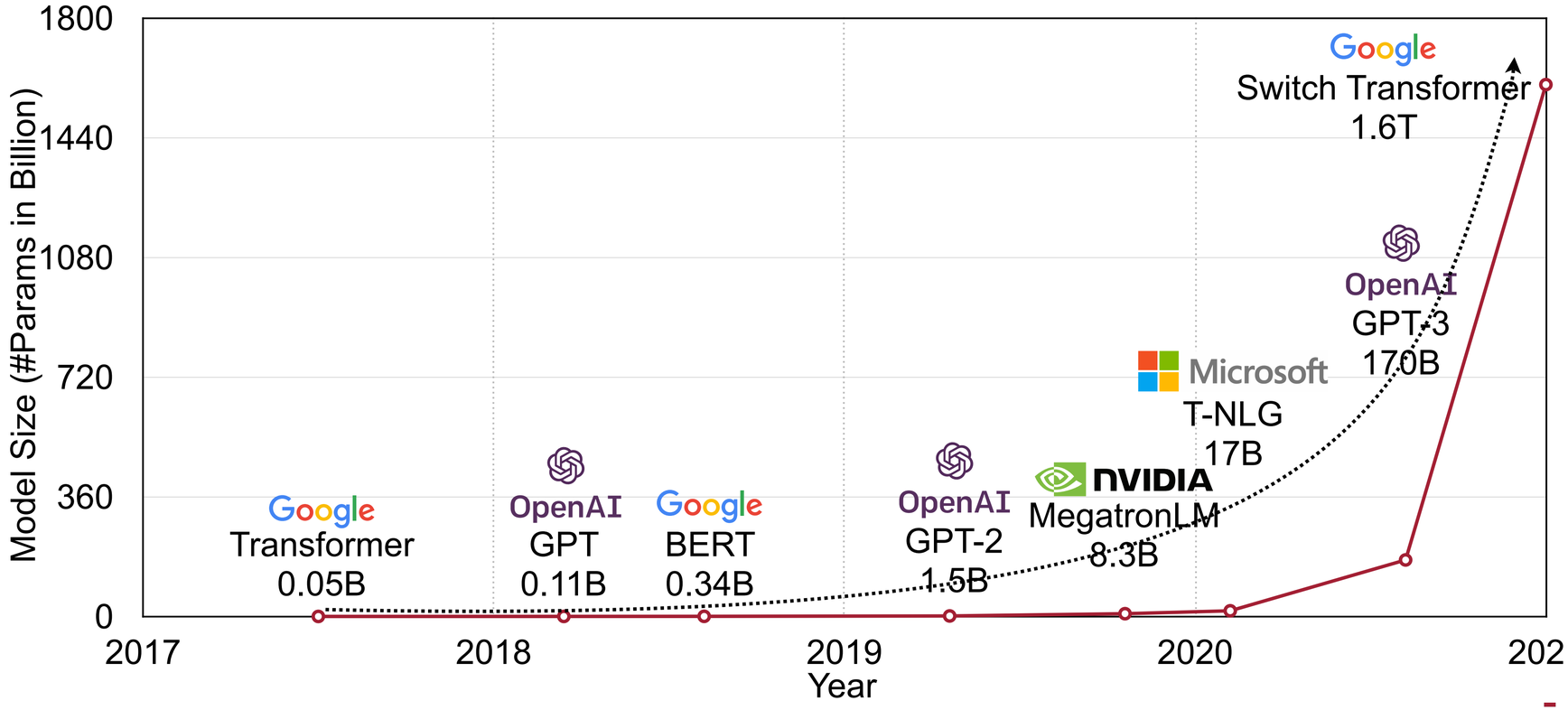
Guangxuan Xiao[*,1], **Ji Lin**[*,1],
Mickael Seznec[2], **Julien Demouth**[2], **Song Han**[1]

[1] **MIT** [2] **NVIDIA**

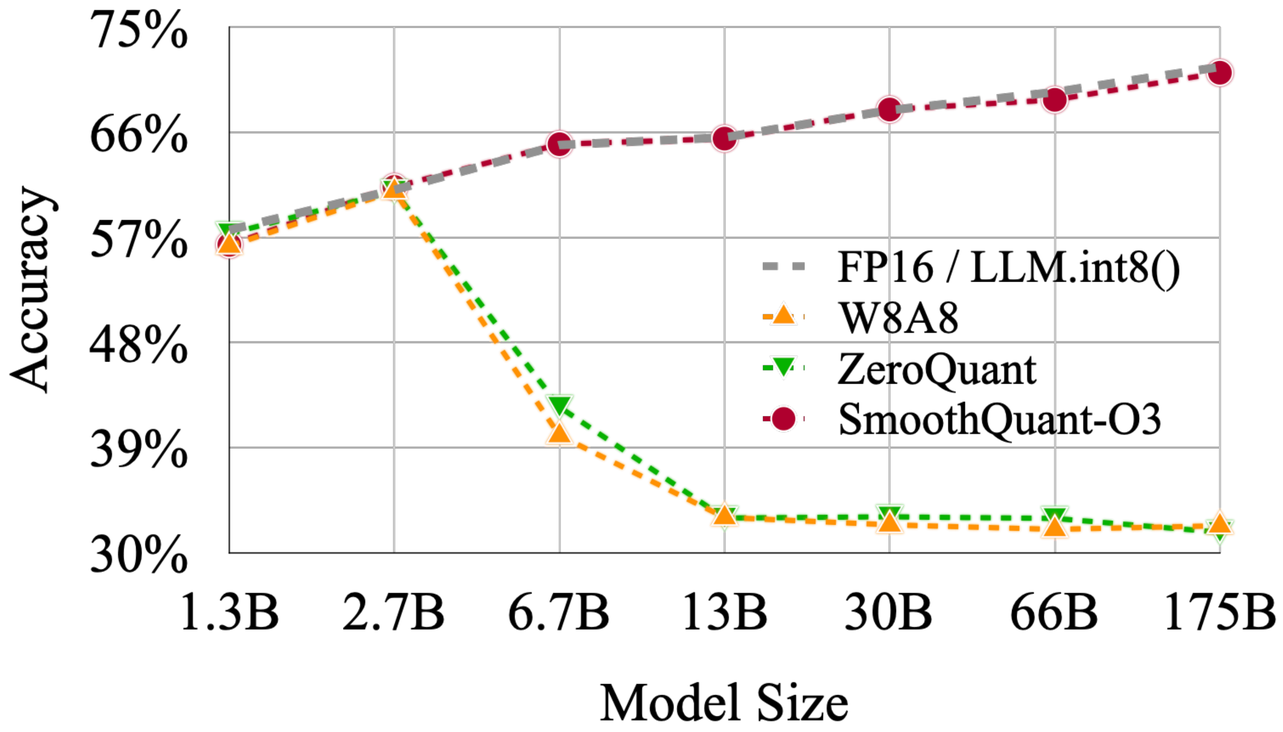
[*] **Equal contribution**

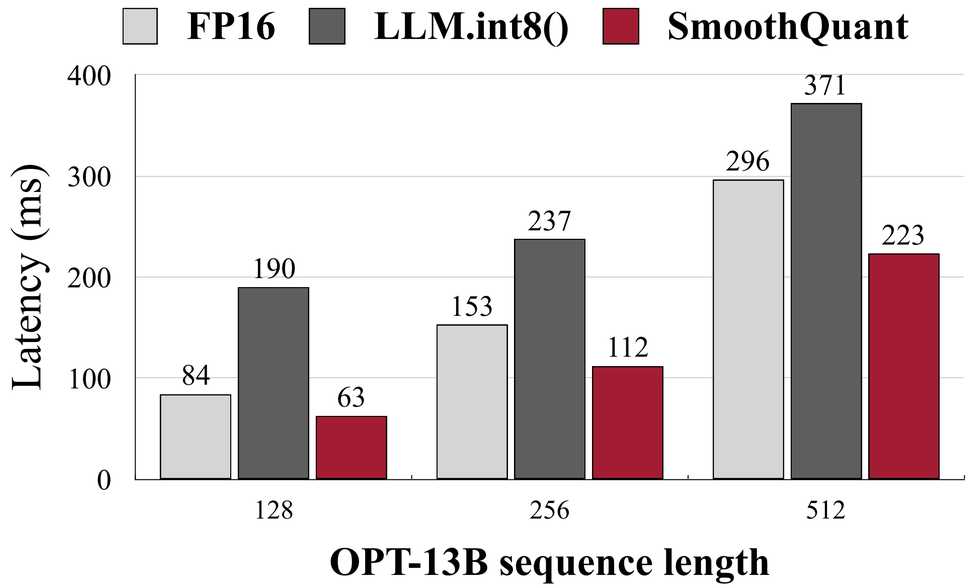
Quantization for LLMs is Important

- NLP model size and computation are increasing exponentially. Model Compression with:
  - Quantization (SmoothQuant) <= today's focus: training-free, model-in & model-out.
  - Token pruning (SpAtten)
  - Neural architecture search (HAT, Lite-Transformer)



Existing Quantization Method is Slow or Inaccurate

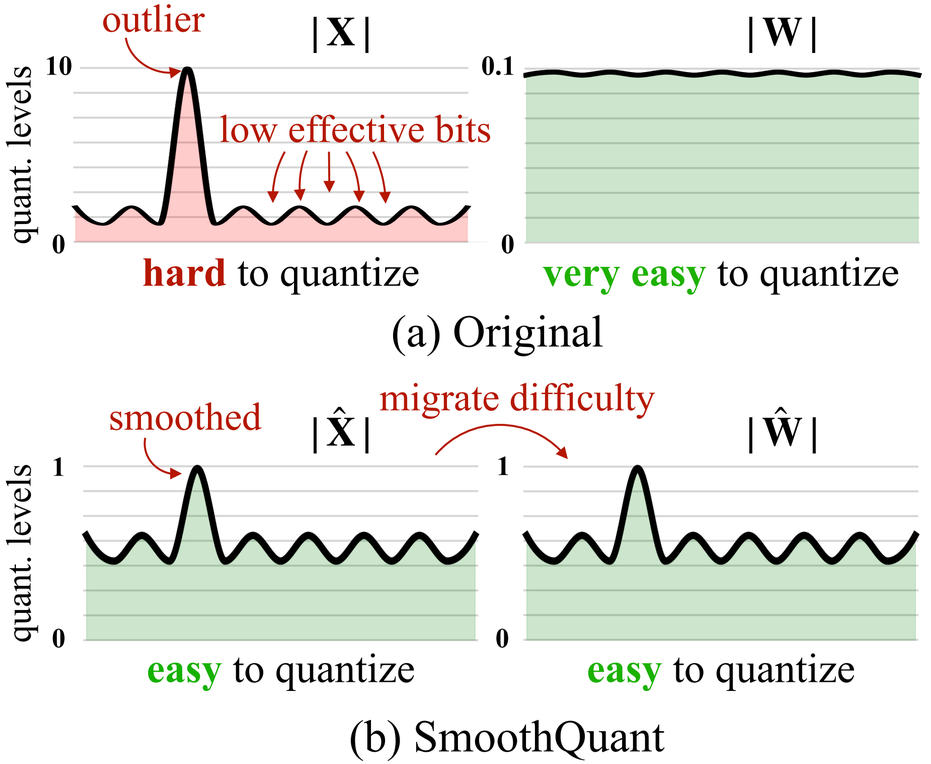




- Systematic outliers emerge in activations when we scale up LLMs beyond 6.7B. Naive but efficient quantization methods will destroy the accuracy.
- The accuracy-preserving baseline, LLM.int8() uses FP16 to represent outliers, which needs runtime outlier detection, scattering and gathering. It is slower than FP16 inference.

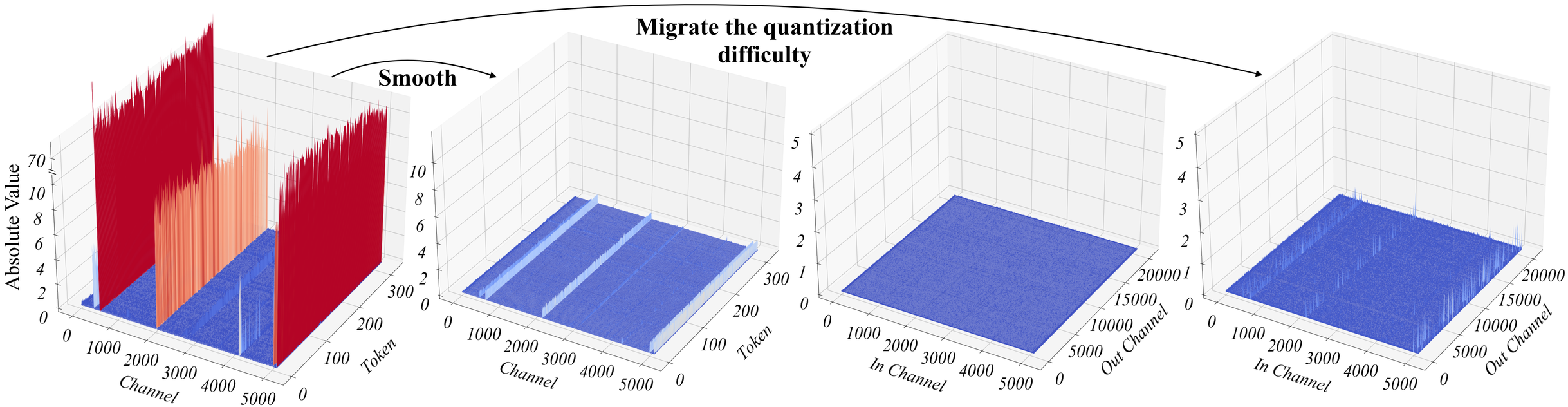
SmoothQuant: Accurate and Efficient Post-Training Quantization for LLMs

	LLM (100B+) Accuracy	Hardware Efficiency
ZeroQuant	✘	✔
Outlier Suppression	✘	✔
LLM.int8()	✔	✘
SmoothQuant	✔	✔



- We propose SmoothQuant, an **accurate** and **efficient** post-training-quantization (PTQ) method to enable 8-bit weight, 8-bit activation (**W8A8**) quantization for LLMs.
- Since **weights are easy** to quantize while **activations are not**, SmoothQuant smooths the activation outliers by **migrating the quantization difficulty from activations to weights** with a mathematically equivalent transformation.

Review the Quantization Difficulty of LLMs



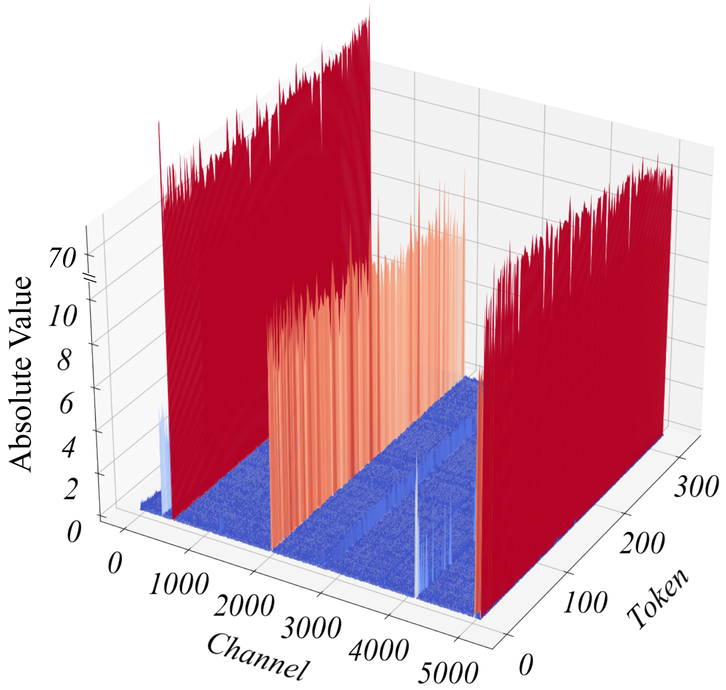
LLMs are notoriously difficult to quantize because:

- Activations are harder to quantize than weights
- Outliers make activation quantization difficult
- Outliers persist in *fixed* channels

Review the Quantization Difficulty of LLMs

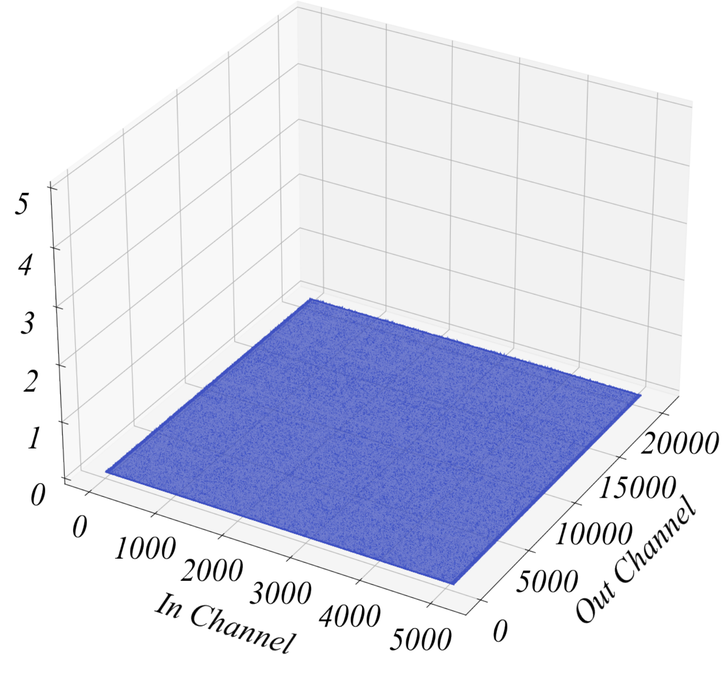
- Activations are harder to quantize than weights

  Previous work has shown quantizing the weights of LLMs with INT8 or even INT4 doesn't degrade accuracy.



Activation (Original)

Hard to quantize



Weight (Original)

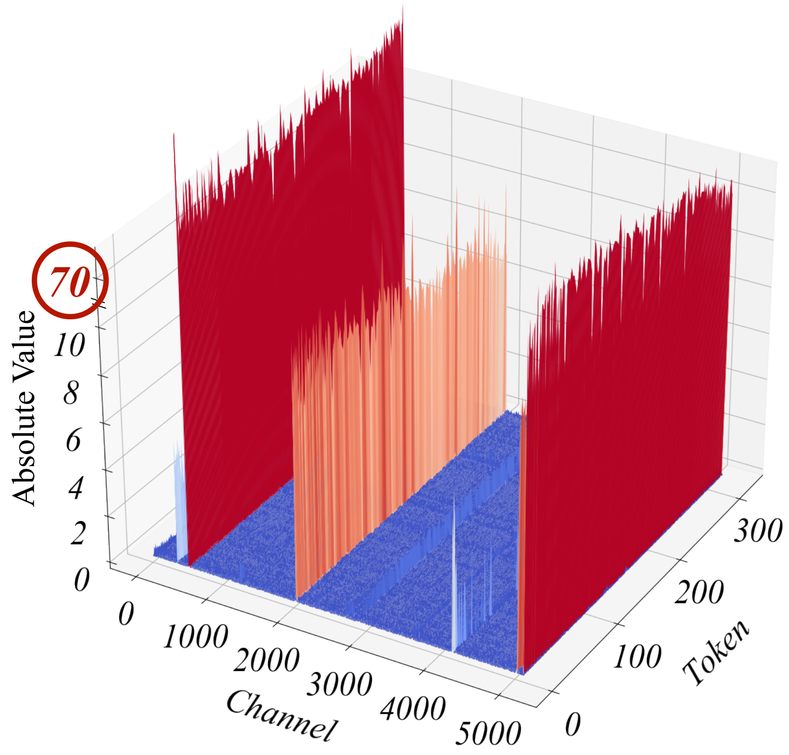
Very easy to quantize

LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale (Dettmers *et al.*, 2022)
GLM-130b: An open bilingual pre-trained model (Zeng *et al.*, 2022)

Review the Quantization Difficulty of LLMs

- Outliers make activation quantization difficult

  The scale of outliers is ~100x larger than most of the activation values.
  If we use INT8 quantization, most values will be zeroed out.



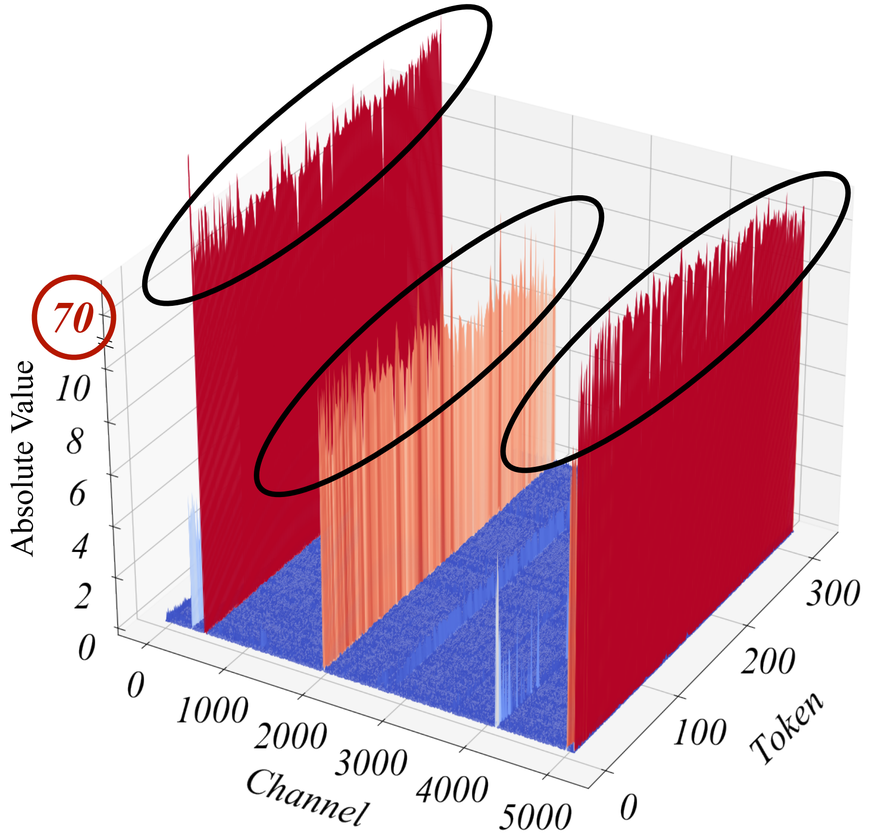
Activation (Original)

Hard to quantize

Review the Quantization Difficulty of LLMs

- Outliers persist in *fixed* channels

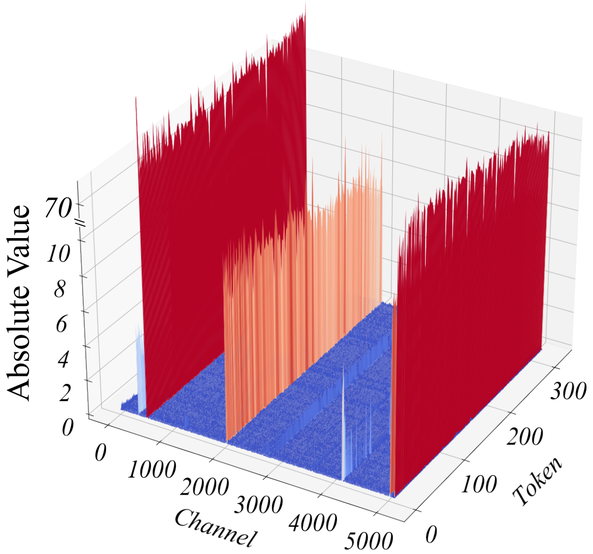
  Fixed channels have outliers, and the outlier channels are persistently large.



Activation (Original)

Hard to quantize

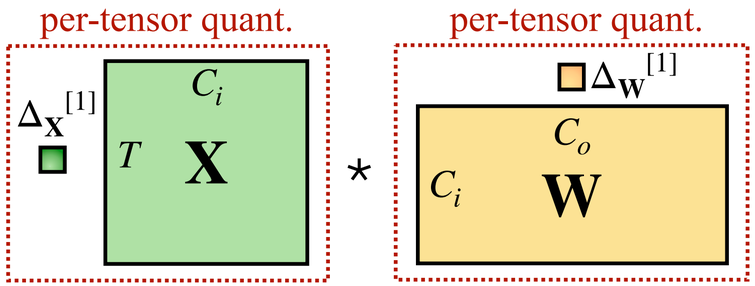
Quantization Schemes



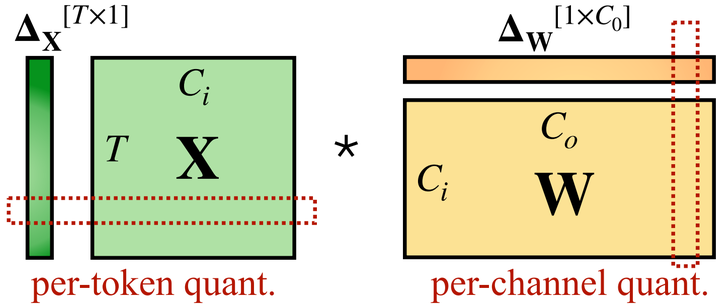
Activation (Original)
Hard to quantize

Model size	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%

Among different activation quantization schemes, only per-channel quantization preserves the accuracy, but it is not compatible with INT8 GEMM kernels.



(a) per-tensor quantization

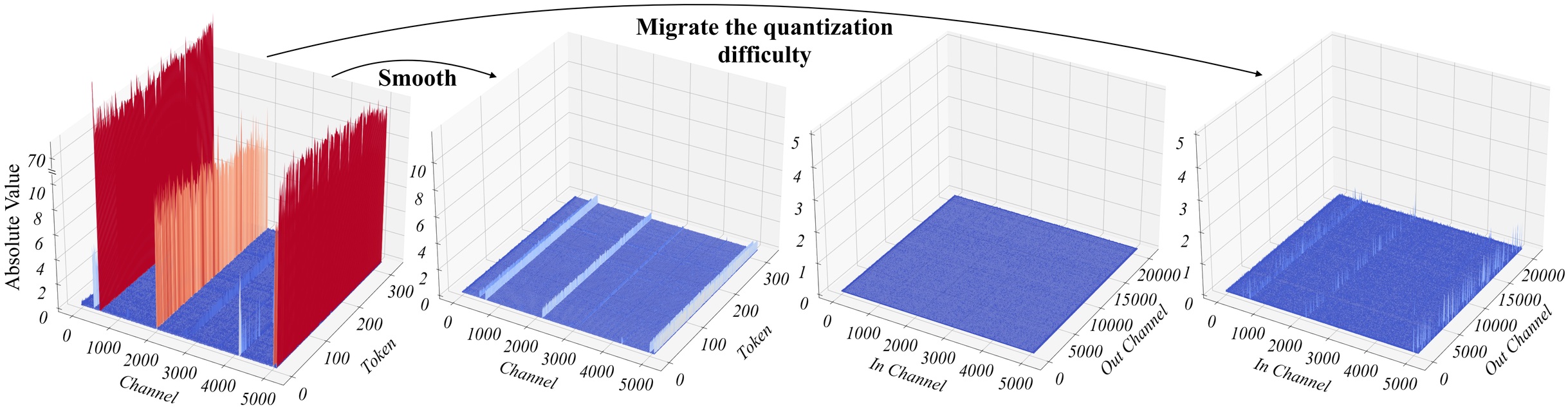


(b) per-token + per-channel quantization

$$\bar{\mathbf{X}}^{\text{INT8}} = \lceil \frac{\mathbf{X}^{\text{FP16}}}{\Delta} \rfloor, \quad \Delta = \frac{\max(|\mathbf{X}|)}{2^{N-1}-1}$$

$$\mathbf{Y} = diag(\mathbf{\Delta}_{\mathbf{X}}^{\text{FP16}}) \cdot (\bar{\mathbf{X}}^{\text{INT8}} \cdot \bar{\mathbf{W}}^{\text{INT8}}) \cdot diag(\mathbf{\Delta}_{\mathbf{W}}^{\text{FP16}})$$

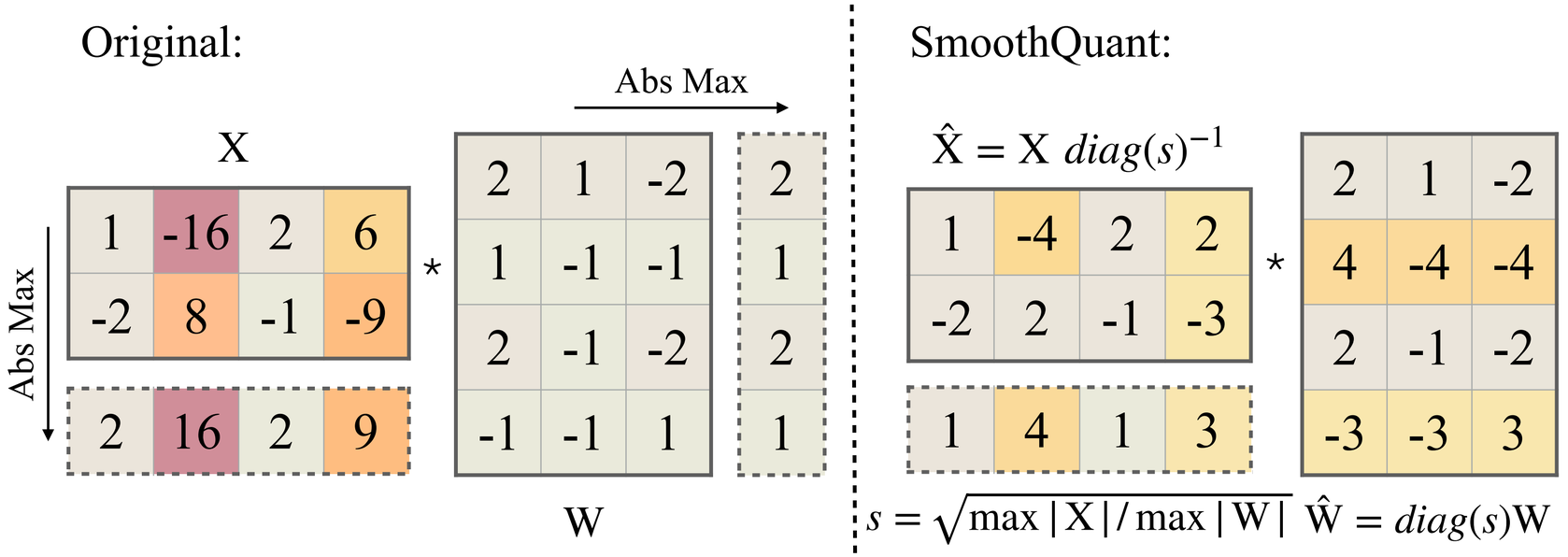
Review the Quantization Difficulty of LLMs



- Activations are harder to quantize than weights
- Outliers make activation quantization difficult
- Outliers persist in *fixed* channels

We can smooth the outlier channels in activations by migrating their magnitudes into the following weights!

Activation Smoothing



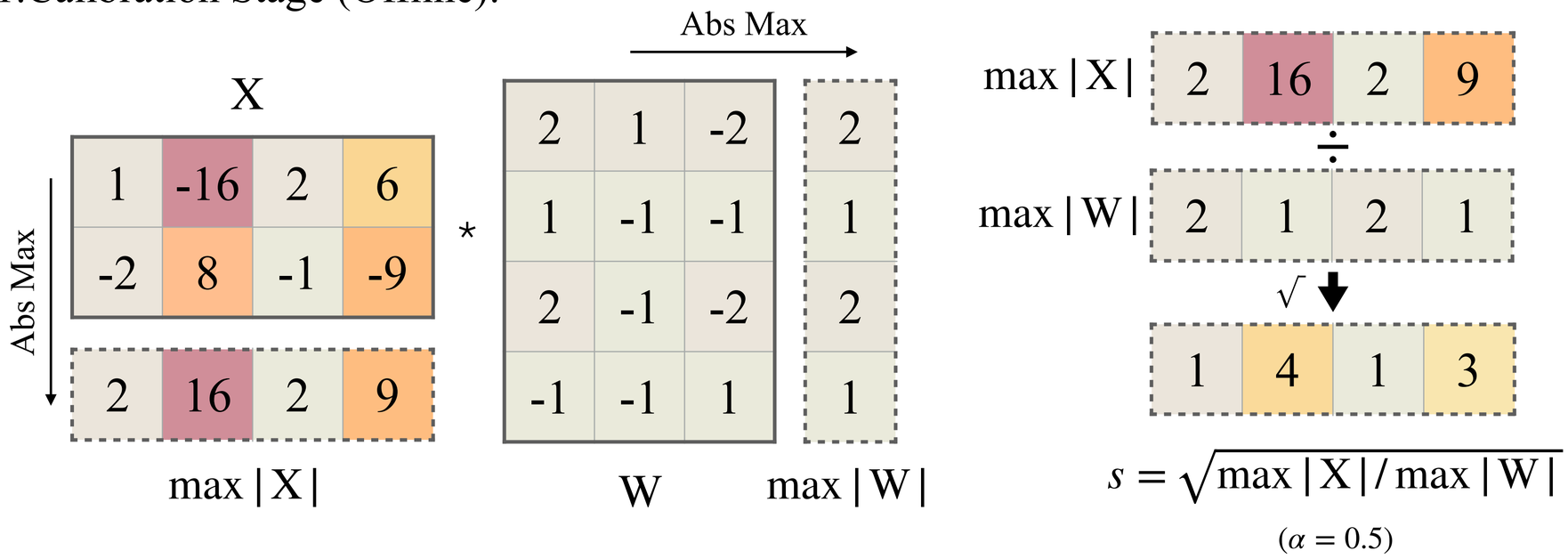
$$\mathbf{s}_j = \max(|\mathbf{X}_j|)^{\alpha} / \max(|\mathbf{W}_j|)^{1-\alpha},\ j = 1,2,\ldots,C_i$$

$$\mathbf{Y} = (\mathbf{X}diag(\mathbf{s})^{-1}) \cdot (diag(\mathbf{s})\mathbf{W}) = \hat{\mathbf{X}}\hat{\mathbf{W}}$$

α: Migration Strength

Activation Smoothing

1.Calibration Stage (Offline):

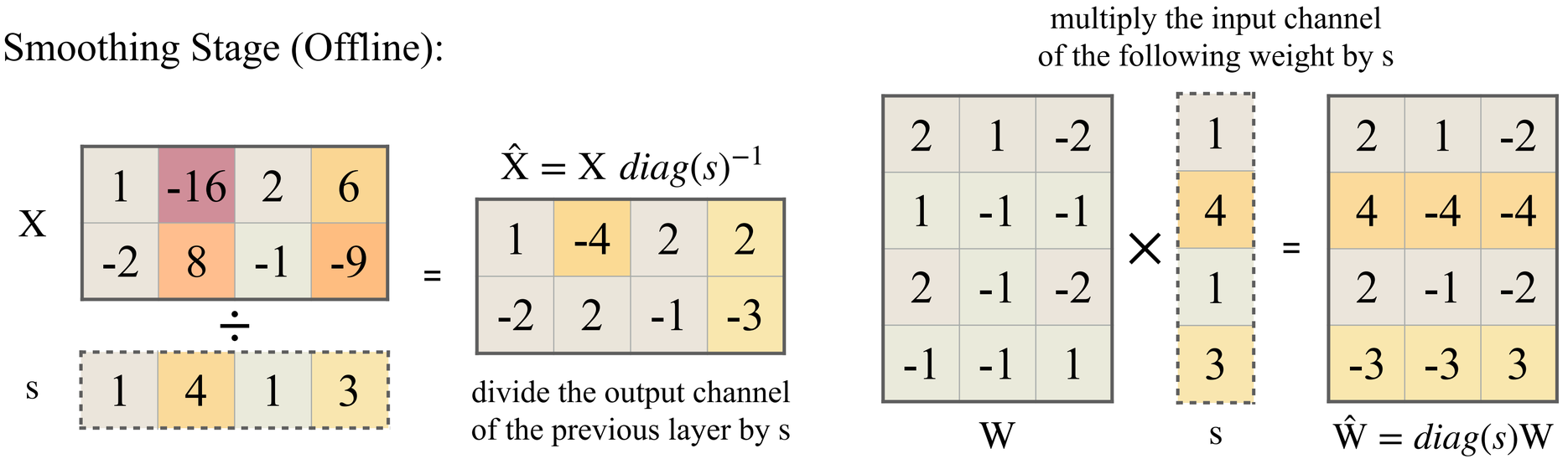


$$\mathbf{s}_j = \max(|\mathbf{X}_j|)^{\alpha} / \max(|\mathbf{W}_j|)^{1-\alpha},\ j = 1,2,\ldots,C_i$$

α: Migration Strength

Activation Smoothing

2. Smoothing Stage (Offline):



$$\mathbf{s}_j = \max(|\mathbf{X}_j|)^{\alpha} / \max(|\mathbf{W}_j|)^{1-\alpha},\ j = 1,2,\ldots,C_i$$

$$\mathbf{Y} = (\mathbf{X}diag(\mathbf{s})^{-1}) \cdot (diag(\mathbf{s})\mathbf{W}) = \hat{\mathbf{X}}\hat{\mathbf{W}}$$

α: Migration Strength

Activation Smoothing

3. Inference (deployed model):

$\hat{X}$

1	-4	2	2
-2	2	-1	-3

At runtime, the activations are smooth and easy to quantize

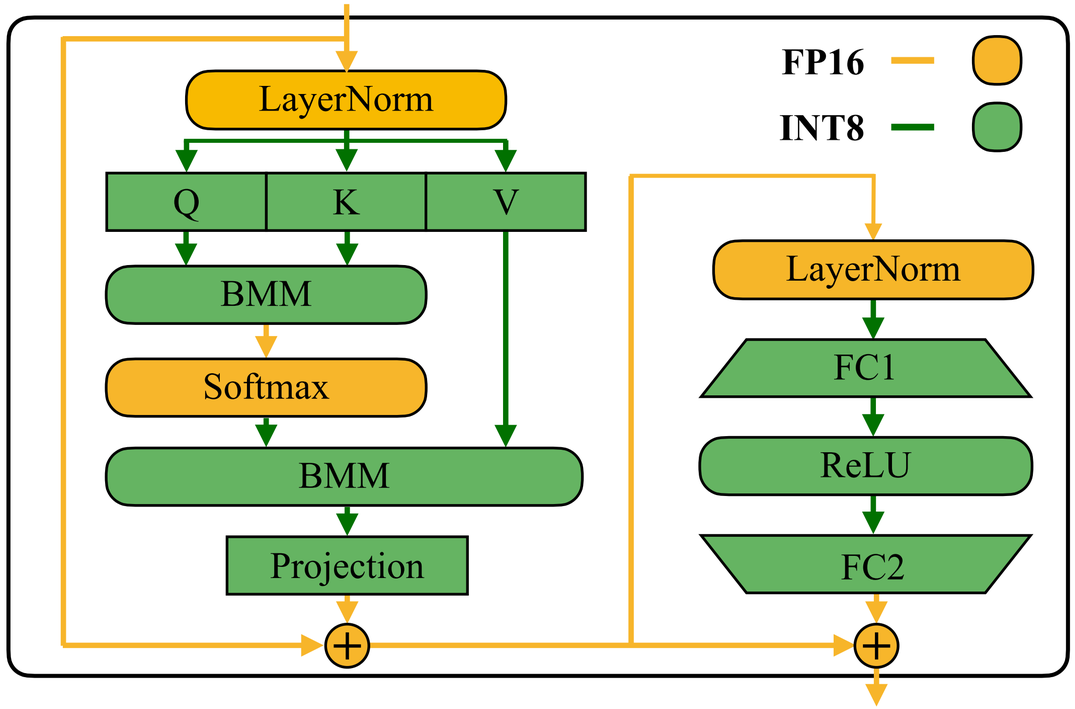
*

2	1	-2
4	-4	-4
2	-1	-2
-3	-3	3

$\hat{W}$

$$\mathbf{Y} = \hat{\mathbf{X}}\hat{\mathbf{W}}$$

System Implementation



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

- SmoothQuant's precision mapping for a Transformer block.
- All compute-intensive operators, such as linear layers and batched matrix multiplications (BMMs) use INT8 arithmetic.

- Quantization setting of the baselines and SmoothQuant. All weight and activations use INT8 representations unless specified.
- We implement three efficiency levels of quantization settings for SmoothQuant. The efficiency improves from O1 to O3.

Accuracy on OPT-175B

OPT-175B	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

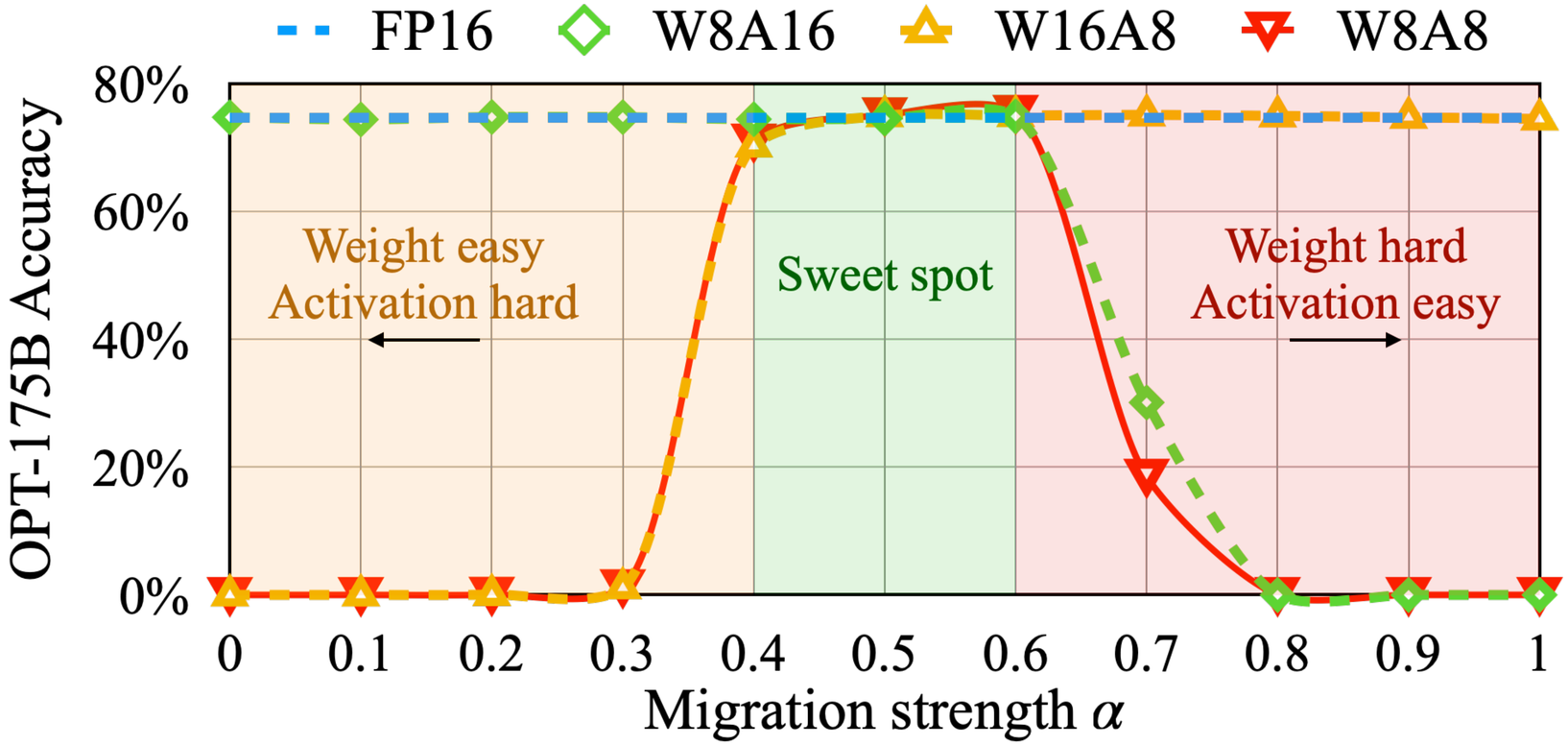
SmoothQuant maintains the accuracy of OPT-175B model after INT8 quantization, even with the most aggressive and most efficient O3 setting.

Accuracy on Different LLMs

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	**71.2%**	68.3%	**73.7%**
SmoothQuant-O2	71.1%	**68.4%**	72.5%
SmoothQuant-O3	71.1%	67.4%	72.8%

SmoothQuant works for different LLMs. We can quantize the 3 largest, openly available LLM models into INT8 without degrading the accuracy.

Ablation Study on the Migration Strength α

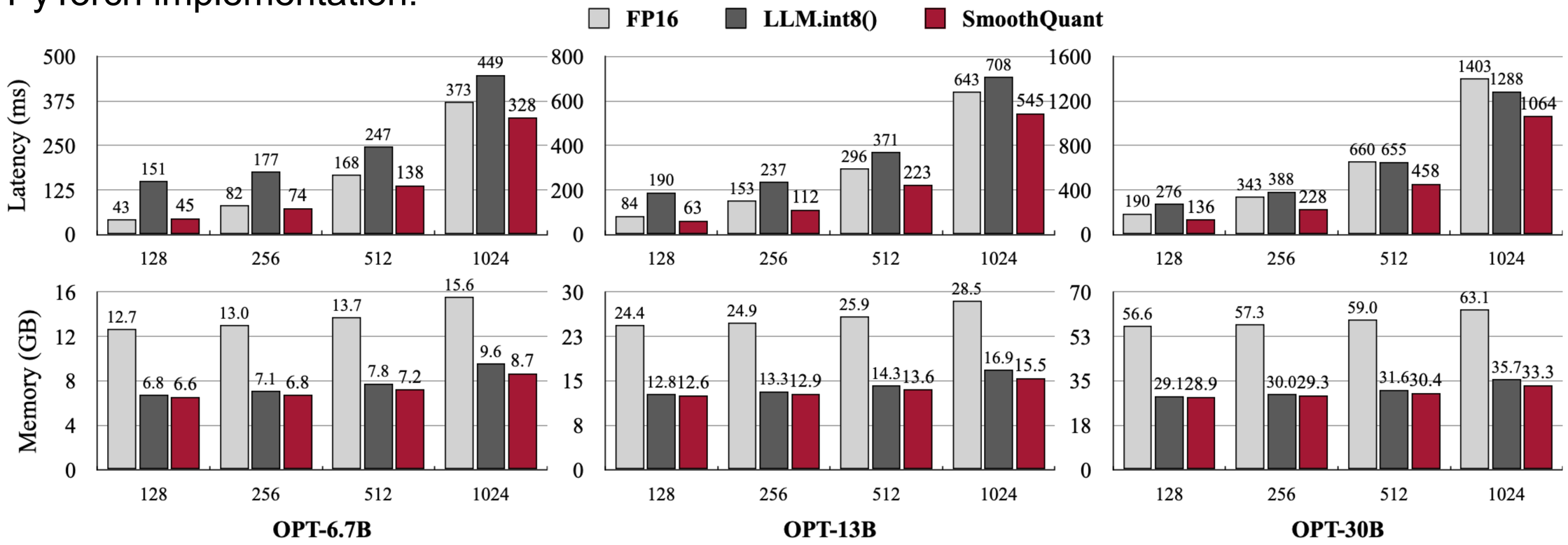


$$\mathbf{s}_j = \max(|\mathbf{X}_j|)^{\alpha} / \max(|\mathbf{W}_j|)^{1-\alpha},\ j = 1,2,\ldots,C_i \qquad \mathbf{Y} = (\mathbf{X}diag(\mathbf{s})^{-1}) \cdot (diag(\mathbf{s})\mathbf{W}) = \hat{\mathbf{X}}\hat{\mathbf{W}}$$

- Migration strength α controls the amount of quantization difficulty migrated from activations to weights.
- A suitable migration strength α (sweet spot) makes both activations and weights easy to quantize.
- If the α is too large, weights will be hard to quantize; if too small, activations will be hard to quantize.

Speedup and Memory Saving

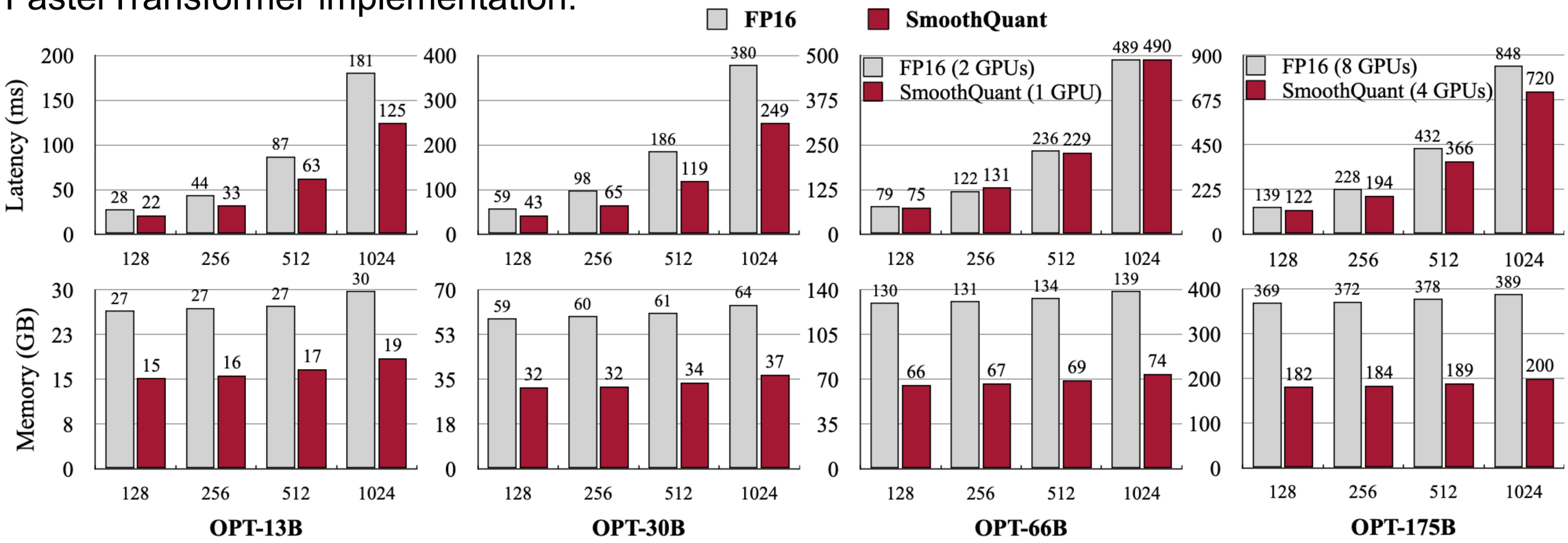
PyTorch implementation:



The PyTorch implementation of SmoothQuant achieves up to **1.51x** speedup and **1.96x** memory saving for OPT models on a single NVIDIA A100-80GB GPU, while **LLM.int8()** slows down the inference in most cases.

Speedup and Memory Saving

FasterTransformer implementation:



- We integrate SmoothQuant into FasterTransformer, a state-of-the-art Transformer serving framework.
- For smaller models, the latency can be significantly reduced with SmoothQuant 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.

Conclusion

- We propose SmoothQuant, a turn-key solution to enable accurate W8A8 quantization for large language models.
- SmoothQuant is accurate and efficient on existing hardware. We can implement SmoothQuant with off-the-shelf kernels to achieve high speedup and memory saving.

- Paper: https://arxiv.org/abs/2211.10438
- Code: https://github.com/mit-han-lab/smoothquant

SpecInfer: Accelerating Generative Large Language Model Serving with Speculative Inference and Token Tree Verification

Xupeng Miao[♠] Gabriele Oliaro[♠] Zhihao Zhang[♠] Xinhao Cheng[♠] Zeyu Wang
Rae Ying Yee Wong Alan Zhu Lijie Yang Xiaoxiang Shi[◇] Chunan Shi[†]
Zhuoming Chen Daiyaan Arfeen Reyna Abhyankar[‡] Zhihao Jia

CMU [◇]*Shanghai Jiao Tong University* [†]*Peking University* [‡]*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

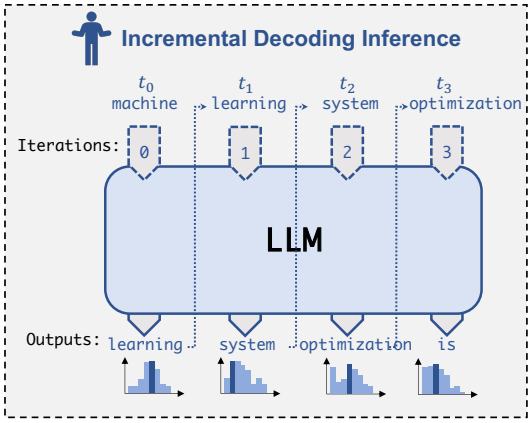
Generative large language models (LLMs), such as ChatGPT [3] and GPT-4 [30], have demonstrated remarkable capabilities of creating natural language texts across various domains, including summarization, instruction following, and question answering [54, 26]. However, it is challenging to quickly and 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].

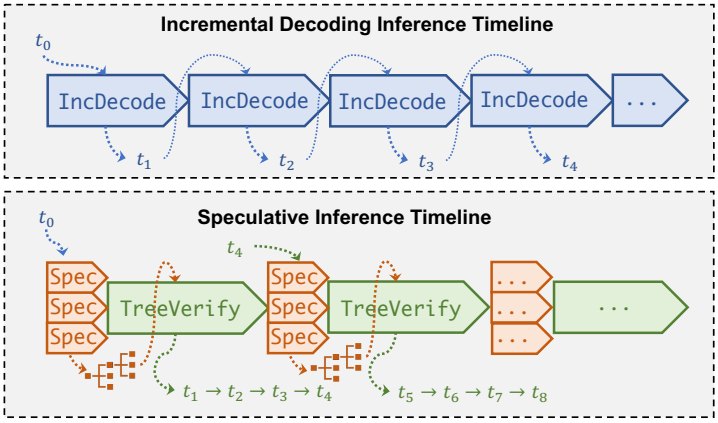
Existing LLM systems generally use an *incremental decoding* approach to serving a request where the system computes the activations for all prompt tokens in a single step and then iteratively decodes *one* new token using the input prompt and all previously generated tokens. This approach respects data 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

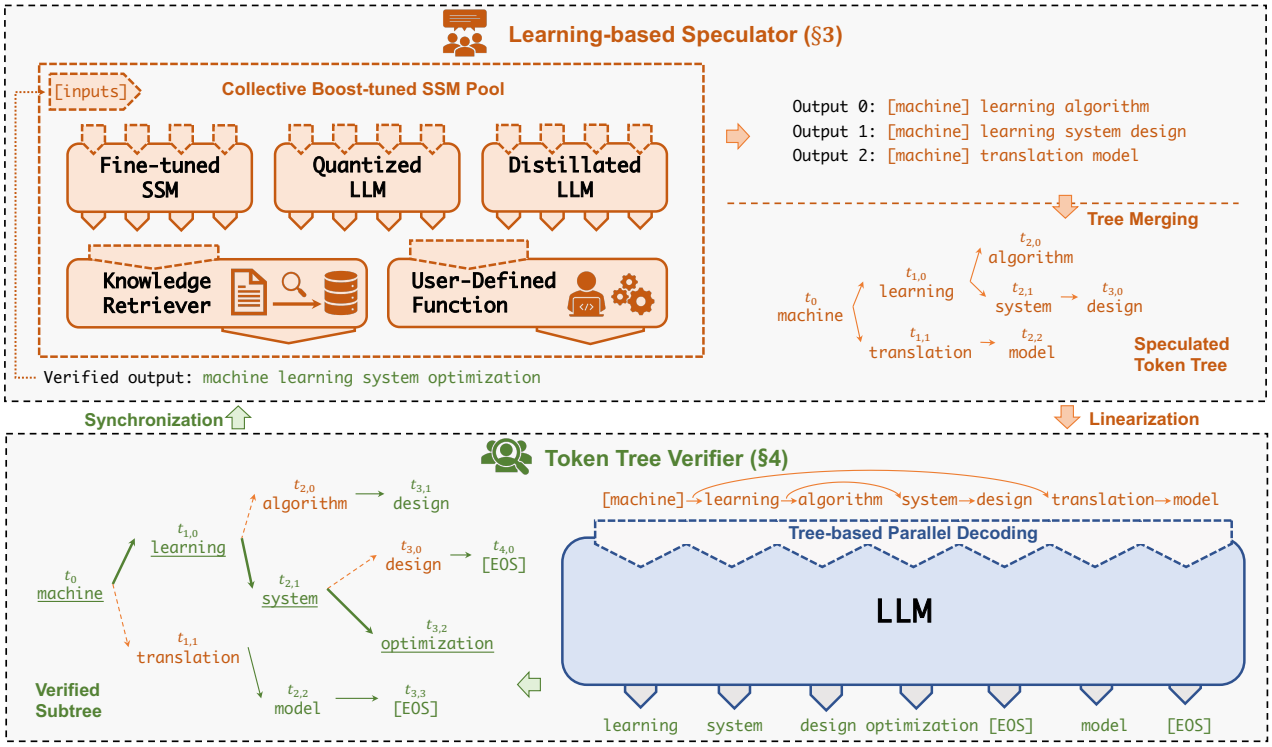
[♠] Contributed equally.



(a) Incremental decoding.



(b) Timeline Comparison.



(c) Speculative inference and token tree verification.

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× 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× for distributed LLM inference and by 2.6-3.5× 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.

Input: A sequence of input tokens $\mathcal{I}$
Output: A sequence of generated tokens
$\mathcal{S} = \mathcal{I}$
while true **do**
  $t = \text{DECODE}(\text{LLM}, \mathcal{S})$
  $\mathcal{S}$.append(t)
  if $t = \langle\text{EOS}\rangle$ **then**
    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.

Input: A sequence of input tokens $\mathcal{I}$
Output: A sequence of generated tokens
$\mathcal{S} = \mathcal{I}$
while true **do**
  $\mathcal{N} = \text{SPECULATE}(\mathcal{S})$
  $\mathcal{O} = \text{TREEPARALLELDECODE}(\text{LLM}, \mathcal{N})$
  if use greedy decoding **then**
    $\mathcal{V} = \text{VERIFYGREEDY}(\mathcal{O}, \mathcal{N})$
  else
    $\mathcal{V} = \text{VERIFYSTOCHASTIC}(\mathcal{O}, \mathcal{N})$
  for $t \in \mathcal{V}$ **do**
    $\mathcal{S}$.append(t)
    if $t = \langle\text{EOS}\rangle$ **then**
      return $\mathcal{S}$

function VERIFYGREEDY($\mathcal{O}, \mathcal{N}$)
  $\mathcal{V} = \emptyset$, $u \leftarrow$ the root of token tree $\mathcal{N}$
  while $\exists v \in \mathcal{N}.p_v = u$ and $t_v = \mathcal{O}(u)$ **do**
    $u = v$
    $\mathcal{V}$.append(t_v)
  $\mathcal{V}$.append($\mathcal{O}(u)$)
  return $\mathcal{V}$

function VERIFYSTOCHASTIC($\mathcal{O}, \mathcal{N}$)
  $\mathcal{V} = \emptyset$, $u \leftarrow$ the root of token tree $\mathcal{N}$
  while u is a non-leaf node **do**
    $\mathcal{H} = \text{child}(u)$ ▷ The set of child nodes for u
    while $\mathcal{H}$ is not empty **do**
      $s \sim \text{rand}(\mathcal{H}), r \sim U(0,1), x_s = \mathcal{H}[s]$
      if $r \le P(x_s \mid u, \Theta_{LLM}) / P(x_s \mid u, \Theta_{SSM_s})$ **then**
        ▷ Token x_s passes verification.
        $\mathcal{V}$.append(x_s)
        $u = s$
        break
      else
        ▷ Normalize the residual $P(x \mid u, \Theta_{LLM})$
        $P(x \mid u, \Theta_{\text{LLM}}) := \text{norm}(\max(0, P(x \mid u, \Theta_{\text{LLM}}) - P(x \mid u, \Theta_{\text{SSM}_s})))$
        $\mathcal{H}$.pop(s)
    if $\mathcal{H}$ is empty **then**
      break
  ▷ All SSMs fail verification; sample the next token
  $x_{\text{next}} \sim P(x \mid u, \Theta_{LLM})$
  $\mathcal{V}$.append(x_{next})
  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

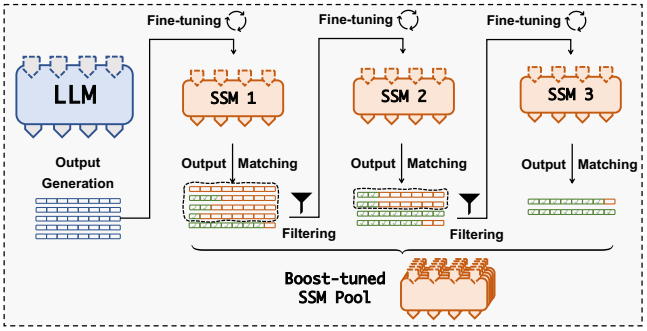


Figure 2: Illustrating SpecInfer's collective boost-tuning technique. When using a single SSM to generate token trees, SpecInfer can verify 2.6 tokens on average in each LLM decoding step. This is due to the misalignment between an SSM and LLM on the first four token sequences. By collectively boost-tuning three SSMs, the average number of verified tokens per LLM decoding step is improved to 7.2.

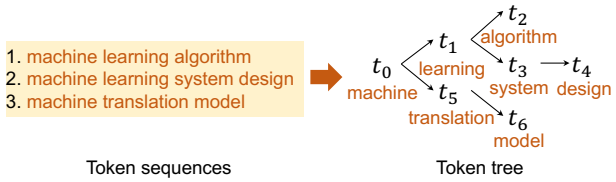


Figure 3: Merging three sequences of tokens (left) into a token tree (right), whose node each identifies a token sequence. For example, node t_4 represents a token sequence $[t_0, t_1, t_3, t_4]$ that translates to "machine learning system design".

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 $\{t_u\}$[1].

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 $\{\mathcal{N}_i\}$ $(1 \le i \le m)$ if and only if $\forall 1 \le i \le m, \forall u \in \mathcal{N}_i, \exists v \in \mathcal{M}$ 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 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.

$$Q_i = X \times W_i^Q, \qquad K_i = X \times W_i^K, \tag{1}$$

$$V_i = X \times W_i^V, \qquad A_i = \frac{(Q_i \times K_i^T)}{\sqrt{d}}, \tag{2}$$

$$H_i = \text{softmax}\big(\text{mask}(A_i)\big)V_i, \quad O = (H_1, ..., H_h)W^O \tag{3}$$

where Q_i, K_i, and V_i denote the query, key, and value tensors of the i-th attention head $(1 \le i \le h)$, 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:

$$\text{mask}(A)_{jk} = \begin{cases} A_{jk} & j \ge k \\ -\infty & j < k \end{cases}. \tag{4}$$

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[2]. 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

[1] For the root node r, S_r represents the token sequence $\{t_r\}$.

[2] 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., $\text{mask}(\text{A})_{\text{jk}} = -\infty$ 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 $t_2, t_3, ..., t_9$ 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 casual 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., $t_3, t_4, ..., t_9$) 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

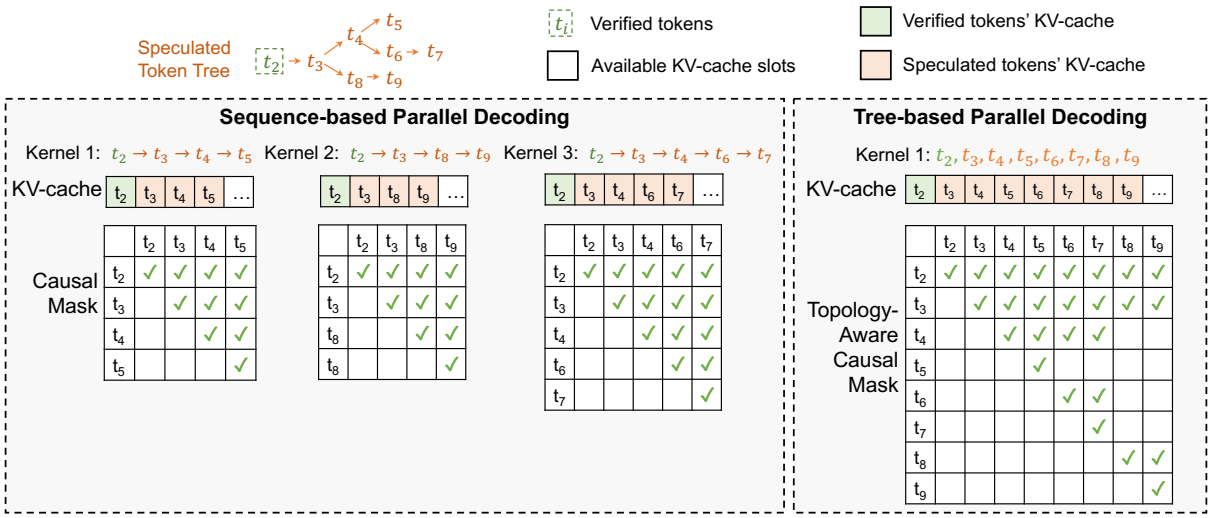


Figure 4: Comparing SpecInfer's tree-based parallel decoding with existing sequence-based decoding.

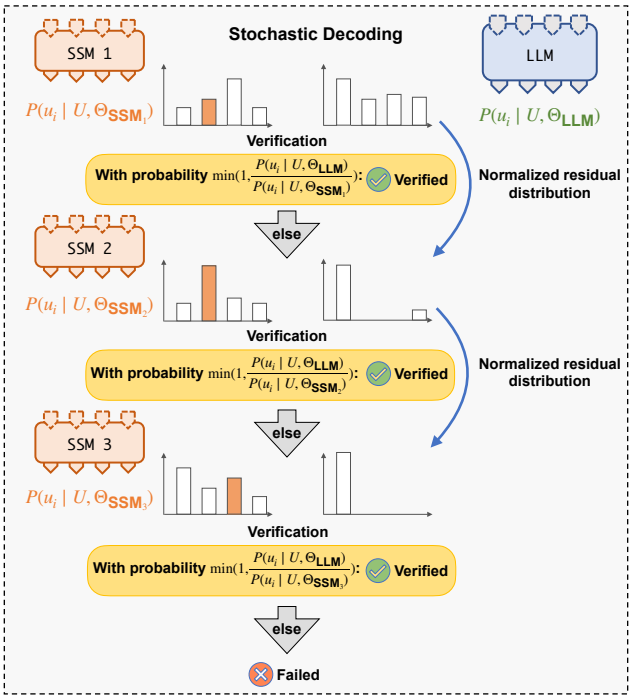


Figure 5: Illustrating SpecInfer's multi-step speculative sampling mechanism for verifying LLMs with stochastic sampling.

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 $P(u_i|u_0,...,u_{i-1};\Theta_{LLM})$, where $U = u_0,...,u_{i-1}$ are previously generated tokens, u_i is the next token to generate, and Θ_{LLM} represents a parameterized LLM.

To verify a speculated token tree with stochastic sampling, we introduce a *multi-step speculative sampling* algorithm to conduct verfication, 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,...,SSM_m, let $P(u_i|U;\Theta_{LLM})$ be the probability distribution of sampling a token using stochastic sampling, where $U = u_0,...,u_{i-1}$ are previously generated tokens, u_i is the next token to generate, Θ_{LLM} represents the parameterized LLM. Let $P_{rmSpecInfer}(u_i|U;\Theta_{LLM},\{\Theta_{SSM_j}\})$ be the probability distribution of sampling token u_i using SpecInfer's multi-step speculative sampling (see the* VERIFYSTOCHASTIC *function in Algorithm 2), where Θ_{SSM_j} is the j-th parameterized SSM. Then $\forall U, u_i, \Theta_{LLM}, \Theta_{SSM_j}$ we have*

$$P(u_i \mid U;\Theta_{LLM}) = P_{\text{SpecInfer}}(u_i \mid U;\Theta_{LLM},\{\Theta_{SSM_j}\}) \quad (6)$$

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 $x \sim P(u_i \mid U;\Theta_{LLM})$ 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 $P\left(reject \mid MSS, U, \Theta_{LLM}, \{\Theta_{SSM_j}\}\right)$ denote the probability of rejecting speculation following multi-step speculative sampling with abbreviation $P(reject \mid MSS)$, and $P\left(reject \mid NS, U, \Theta_{LLM}, \{\Theta_{SSM_j}\}\right)$ the probability of rejecting speculation following Naive Sampling (NS) with abbreviation $P(reject \mid NS)$. Then $\forall U, \Theta_{LLM}, \{\Theta_{SSM_j}\}$, we have*

$$P(reject \mid MSS) \leq P(reject \mid NS)$$

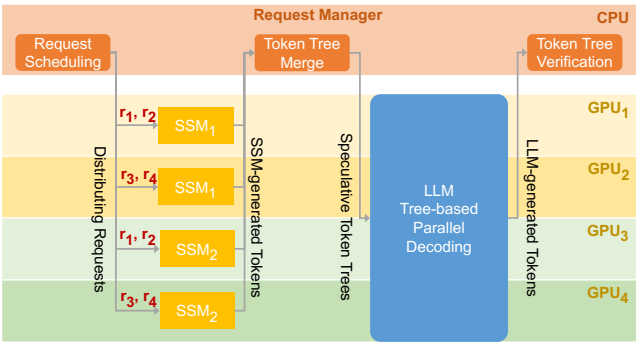


Figure 6: SpecInfer's workflow for one iteration of speculative inference and token tree verification. SpecInfer uses data parallelism to serve SSMs, and combine tensor model parallelism and pipeline model parallelism for serving an LLM.

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 serves 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 cuTLASS) to execute inference computation on GPUs. With speculative inference and token tree

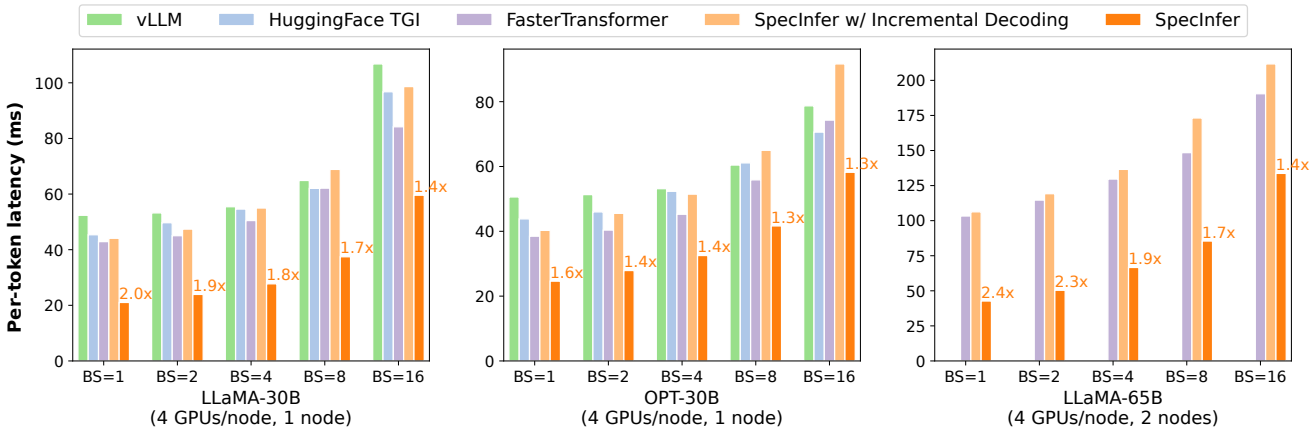


Figure 7: Comparing the end-to-end inference latency of incremental decoding and SpecInfer on five prompt datasets. We use LLaMA-7B as the LLM and all SSMs are derived from LLaMA-160M. The performance is normalized by incremental decoding, and the numbers on the SpecInfer bars indicate the speedups over incremental decoding.

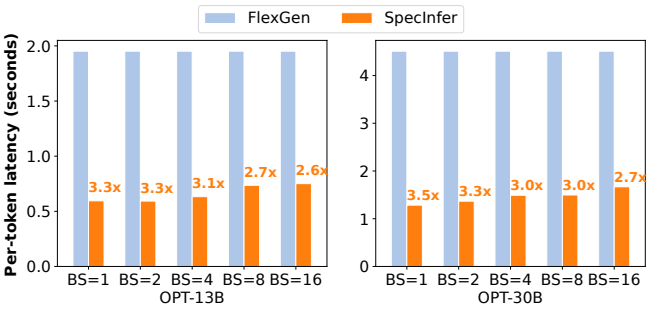


Figure 8: Comparing the end-to-end offloading-based inference latency of FlexGen and SpecInfer. Both FlexGen and SpecInfer perform model offloading to serve OPT-13B and OPT-30B models on a single 24GB A10 GPU.

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 SSMs, all of which were derived from LLaMA-160M. The speculation length is 16 for all SSMs.

# SSMs	1	2	3	4	5
CIP	3.35	3.74	3.97	4.05	**4.11**
CP	2.71	3.14	3.32	3.45	**3.51**
WebQA	2.84	3.08	3.20	3.27	**3.31**
Alpaca	2.70	3.19	3.36	3.44	**3.49**
PIQA	2.98	3.21	3.36	3.44	**3.49**
Avg	2.92	3.27	3.44	3.53	**3.58**

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.

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	**3.74**
CP	2.95	3.35	3.49	3.52	**3.68**
WebQA	2.51	2.92	3.04	3.09	**3.20**
Alpaca	3.33	3.89	4.06	4.17	**4.35**
PIQA	2.75	3.14	3.26	3.31	**3.43**
Avg	2.91	3.34	3.47	3.53	**3.68**

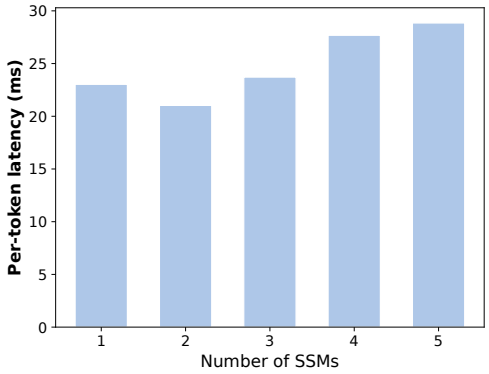


Figure 9: Comparing SpecInfer's performance with different numbers of SSMs. We use LLaMA-30B as the LLM and the five collectively boost-tuned LLaMA-160M models as the SSMs.

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. Piqa: 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 Hoefler, 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 Hennessy 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, 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, Mostofa 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, 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.

[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, 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 Efrain 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, Lukasz 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, Daxin 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, Shuohui 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,...,SSM_m, let $P(u_i|U;\Theta_{LLM})$ be the probability distribution of sampling a token using stochastic sampling, where $U = u_0,...,u_{i-1}$ are previously generated tokens, u_i is the next token to generate, Θ_{LLM} represents the parameterized LLM. Let $P_{rmSpecInfer}(u_i|U;\Theta_{LLM},\{\Theta_{SSM_j}\})$ be the probability distribution of sampling token u_i using SpecInfer's multi-step speculative sampling (see the* VERIFYSTOCHASTIC *function in Algorithm 2), where Θ_{SSM_j} is the j-th parameterized SSM. Then $\forall U, u_i, \Theta_{LLM}, \Theta_{SSM_j}$ we have*

$$P(u_i \mid U;\Theta_{LLM}) = P_{\text{SpecInfer}}(u_i \mid U;\Theta_{LLM},\{\Theta_{SSM_j}\}) \tag{7}$$

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 $r_j = \sum_i \max\left(0, P(u_i \mid U,\Theta_{\text{SSM}_j}) - P(u_i \mid U,\Theta_{\text{LLM}})\right), \forall j \in [m]$ as the rejection probability at round j, $T_0 = P(u \mid U,\Theta_{\text{LLM}})$ and $T_j = \frac{T_{j-1} - P(u|U,\Theta_{\text{SSM}_j})}{r_j}, \forall j \in [m]$. We further define

$$A_m = \max(0, T_m),$$

$$A_{j-1} = \min(P(u \mid U,\Theta_{\text{SSM}_j}), \max(T_{j-1}, 0)) + r_j A_j, \forall j \in [m]$$

Then following the VERIFYSTOCHASTIC sampling procedure stated in Algorithm 2, the probability of sampling u_i is:

$$P_{\text{SpecInfer}}(u_i \mid U;\Theta_{LLM},\{\Theta_{SSM_j}\}) = A_0$$

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:

$$\begin{aligned}
A_{n-1} &= \min(P(u \mid U,\Theta_{\text{SSM}_n}), \max(T_{n-1}, 0)) + r_n A_n \\
&= \min(P(u \mid U,\Theta_{\text{SSM}_n}), \max(T_{n-1}, 0)) + r_n \max(0, T_n) \\
&= \min(P(u \mid U,\Theta_{\text{SSM}_n}), \max(T_{n-1}, 0)) \\
&+ \max(0, T_{n-1} - P(u \mid U,\Theta_{\text{SSM}_n})) \\
&= \max(0, T_{n-1})
\end{aligned}$$

Thus, we have $A_j = \max(0, T_j), \forall j \in [m]$. Combining previous results, we have $P_{\text{SpecInfer}}(u_i \mid U;\Theta_{LLM},\{\Theta_{SSM_j}\}) = A_0 = \max(0, T_0) = P(u_i \mid U;\Theta_{LLM})$, which concludes our proof. Notice that the overall rejection probability is $\prod_{j=1}^m r_j$. □

Theorem A.2. *Let $P\left(\text{reject} \mid MSS, U, \Theta_{LLM}, \{\Theta_{SSM_j}\}\right)$ denote the probability of rejecting speculation following multi-step speculative sampling with abbreviation $P(\text{reject} \mid MSS)$, and $P\left(\text{reject} \mid NS, U, \Theta_{LLM}, \{\Theta_{SSM_j}\}\right)$ the probability of rejecting speculation following Naive Sampling (NS) with abbreviation $P(\text{reject} \mid NS)$. Then $\forall U, \Theta_{LLM}, \{\Theta_{SSM_j}\}$, we have*

$$P(\text{reject} \mid MSS) \le P(\text{reject} \mid NS)$$

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

$$P(\text{reject} \cap \mathcal{O}(u) = t \mid \text{NS}) \ge P(\text{reject} \cap \mathcal{O}(u) = t \mid \text{MSS})$$

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 $P(t \mid U,\Theta_{\text{LLM}}) \prod_{j=1}^m (1 - P(t \mid U,\Theta_{\text{SSM}_j}))$ 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 $A_0 = P(t \mid U,\Theta_{\text{LLM}})$ and $A_j = \frac{A_{j-1} - \min(P(t|U,\Theta_{\text{SSM}_j}), A_{j-1})}{r_j} = \max(0, \frac{A_{j-1} - P(t|U,\Theta_{\text{SSM}_j})}{r_j}), \forall j \in [m]$. 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 $\max(0, (P(t \mid U,\Theta_{\text{LLM}}) - P(t \mid U,\Theta_{\text{SSM}_1})) \prod_{j=2}^m (1 - P(t \mid U,\Theta_{\text{SSM}_j})))$. We now case on whether there exists round k such that $P(t \mid U,\Theta_{\text{SSM}_k}) \ge A_{k-1}$.

Case 1: $P(t \mid U,\Theta_{\text{SSM}_k}) \ge A_{k-1}$ for some k. Then $A_\ell = 0$ for all $\ell \ge k$, and $A_m \prod_{j=1}^m r_j = 0$, and our upper bound holds.

Case 2: $P(t \mid U,\Theta_{\text{SSM}_k}) < A_{k-1}$ for all rounds k. Then $A_j = \frac{A_{j-1} - P(t|U,\Theta_{\text{SSM}_j})}{r_j}, \forall j \in [m]$. A lower bound for r_j is $A_{j-1} - P(t \mid U,\Theta_{\text{SSM}_j})$. We show

$A_m \prod_{j=1}^m r_j \le (P(t \mid U,\Theta_{LLM}) - P(t \mid U,\Theta_{\text{SSM}_1})) \prod_{j=2}^m (1 - P(t \mid U,\Theta_{\text{SSM}_j}))$

To do this, we prove the stronger claim that

$A_n \prod_{j=1}^n r_j \le (P(t \mid U,\Theta_{LLM}) - P(t \mid U,\Theta_{\text{SSM}_1})) \prod_{j=2}^n (1 - P(t \mid U,\Theta_{\text{SSM}_j}))$

for all $n \le m$ through induction.

BC: $n = 1$. As $A_1 = \frac{A_0 - P(t|U,\Theta_{\text{SSM}_1})}{r_1} = \frac{P(t|U,\Theta_{LLM}) - P(t|U,\Theta_{\text{SSM}_1})}{r_1}$, the LHS is $p(t|u) - P(t \mid U,\Theta_{\text{SSM}_1})$, which is equal to the RHS.

IH: Assume the upper bound holds for all $n < N$.

IS: Consider the case where $n = N$. Then

$$A_N = \frac{A_{N-1} - P(t \mid U,\Theta_{\text{SSM}_N})}{r_N}$$

The LHS is

$$
\begin{aligned}
&\frac{A_{N-1} - P(t \mid U, \Theta_{\text{SSM}_N})}{r_N} \prod_{j=1}^{N} r_j \\
&= (A_{N-1} - P(t \mid U, \Theta_{\text{SSM}_N})) \prod_{j=1}^{N-1} r_j \\
&= \left(A_{N-1} \prod_{j=1}^{N-1} r_j \right) - \left(P(t \mid U, \Theta_{\text{SSM}_N}) \prod_{j=1}^{N-1} r_j \right) \\
&\leq \left(A_{N-1} \prod_{j=1}^{N-1} r_j \right) - P(t \mid U, \Theta_{\text{SSM}_N}) \left(A_{N-1} \prod_{j=1}^{N-1} r_j \right) \\
&\qquad\qquad (0 \leq A_{N-1} \leq 1 \text{ and all factors non-negative}) \\
&= (1 - P(t \mid U, \Theta_{\text{SSM}_N})) \left(A_{N-1} \prod_{j=1}^{N-1} r_j \right) \\
&\leq (1 - P(t \mid U, \Theta_{\text{SSM}_N}))(P(t \mid U, \Theta_{LLM}) - P(t \mid U, \Theta_{\text{SSM}_1})) \\
&\quad \prod_{j=1}^{N-1} (1 - P(t \mid U, \Theta_{\text{SSM}_j})) \qquad \text{(IH)} \\
&= (P(t \mid U, \Theta_{LLM}) - P(t \mid U, \Theta_{\text{SSM}_1})) \prod_{j=1}^{N} (1 - P(t \mid U, \Theta_{\text{SSM}_j}))
\end{aligned}
$$

which is the RHS, deriving the upper bound as desired.

Thus in all cases, $P(\text{reject} \cap \mathcal{O}(u) = t \mid \text{MSS}) \leq \max(0, (P(t \mid U, \Theta_{LLM}) - P(t \mid U, \Theta_{\text{SSM}_1})) \prod_{i=2}^{k} (1 - q_i(t)))$.

We now show $P(\text{reject} \cap \mathcal{O}(u) = t \mid \text{NS}) \geq P(\text{reject} \cap \mathcal{O}(u) = t \mid \text{MSS})$. Note that $P(\text{reject} \cap \mathcal{O}(u) = t \mid \text{NS}) \geq 0$ always, so we only need to compare against $(P(t \mid U, \Theta_{LLM}) - P(t \mid U, \Theta_{\text{SSM}_1})) \prod_{j=2}^{m} (1 - P(t \mid U, \Theta_{\text{SSM}_j}))$ for the cases where $P(t \mid U, \Theta_{\text{SSM}_k}) < A_{k-1}$ for all k; in all other cases $P(\text{reject} \cap \mathcal{O}(u) = t \mid \text{MSS}) = 0$ and the inequality is trivially true. In the cases we are interested in, $P(t \mid U, \Theta_{\text{SSM}_1}) < A_0 = P(t \mid U, \Theta_{LLM})$. Then,

$$
\begin{aligned}
&P(\text{reject} \cap \mathcal{O}(u) = t \mid \text{NS}) \\
&= P(t \mid U, \Theta_{LLM}) \prod_{j=1}^{m} (1 - P(t \mid U, \Theta_{\text{SSM}_j})) \\
&= (P(t \mid U, \Theta_{LLM}) - P(t \mid U, \Theta_{LLM}) P(t \mid U, \Theta_{\text{SSM}_1})) \\
&\quad \prod_{j=2}^{m} (1 - P(t \mid U, \Theta_{\text{SSM}_j})) \\
&\geq (P(t \mid U, \Theta_{LLM}) - P(t \mid U, \Theta_{\text{SSM}_1})) \prod_{j=2}^{m} (1 - P(t \mid U, \Theta_{\text{SSM}_j})) \\
&\qquad\qquad (P(t \mid U, \Theta_{LLM}) \leq 1 \text{ and all factors non-negative}) \\
&\geq P(\text{reject} \cap \mathcal{O}(u) = t \mid \text{MSS})
\end{aligned}
$$

Since the inequality holds for all t in the vocabulary, by the law of total probability, we have $P(\text{reject} \mid \text{MSS}) \leq P(\text{reject} \mid \text{NS})$. □

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[1] 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[2] and CUB [3] 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.

[1] https://github.com/NVIDIA/FasterTransformer
[2] https://github.com/NVIDIA/cutlass
[3] 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 FasterTransfomer, 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 [4] 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.

[4] https://github.com/triton-inference-server/server

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

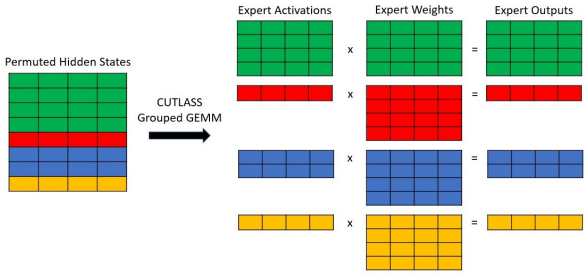


Figure 1: Shows the computation performed by CUTLASS Grouped GEMM. Each color is a sub-matrix for a particular expert, with the matrix multiplies for each expert happening in parallel. If the yellow sentence was finished, it would be omitted from the computation with batch-pruning enabled. This would 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 [5]

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

$W_{dq} \leftarrow$ `NewMatrix`(E, M, N)
for $e \leftarrow 0$ **to** $E - 1$ **do**
  for $m \leftarrow 0$ **to** $M - 1$ **do**
    for $n \leftarrow 0$ **to** $N - 1$ **do**
      $f =$ `IntToFloat`$(W[e, m, n])$
      $W_{dq}[e, m, n] = f * S[e, n]$
    end for
  end for
end for
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

[5] 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.

1. For any FP16 number X where $1024 \leq X < 2048$, 1024 will be represented exactly in the exponent bits and $int(X - 1024)$ will be directly stored in the mantissa. For example, FP16 representation of 1027 (represented as 0x6403) has the integer 3 stored directly in the mantissa bits of its representation.

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 in the FP16 mantissa. This is easily done by performing 0x6400 | Y, since 0x6400 is the hex representation of 1024 in FP16.

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 quanitzed 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 ($\sim$ 40K 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 Δ 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 FasterTransformer 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, Minjia 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, 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. 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

{zheweiyao, 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_{6B} and GPT-$NeoX_{20B}$, 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 $\text{BERT}_{\text{base}}$/$\text{GPT-3}_{\text{350M}}$ 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_{6B} and $\text{GPT-NeoX}_{\text{20B}}$, with INT8 quantization. ZeroQuant can achieve 3.67x speedup over the FP16 model for GPT-J_{6B} and (2) reduce the GPU requirement for inference from 2 to 1 and latency from 65ms to 25ms for $\text{GPT-NeoX}_{\text{20B}}$ (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$_{20B}$) 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$_{base}$ 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.

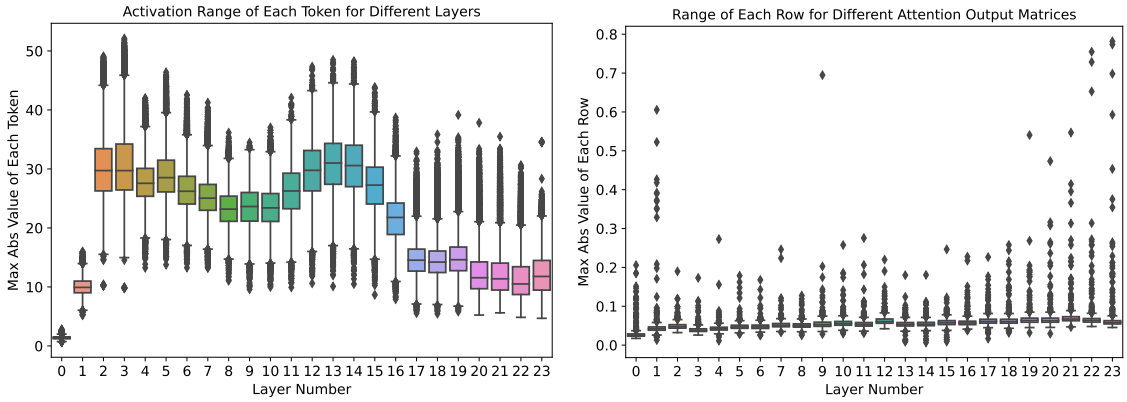


Figure 1: The activation range (left) and row-wise weight range of the attention output matrix (right) of different layers on the pretrained GPT-3$_{350M}$. See Figure C.1 for the results of BERT$_{base}$.

The results of GPT-3$_{350M}$ with PTQ are shown in Table 1. As can be seen, the INT8 activation

Table 1: Post training quantization results of GPT-3_{350M} on 20 zero-shot evaluation datesets. 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_{350M} still achieves reasonable performance like OpenBookQA but it loses accuracy on the majority of the rest tasks. Particularly, for Wikitext-2, GPT-3_{350M} 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_{350M} 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 ($\boldsymbol{W}_o$) of GPT-3_{350M} 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 in to 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_{350M} and $\text{BERT}_{\text{base}}$. 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, ..., L_N, the accessible dataset has input ($\boldsymbol{X}$, $\boldsymbol{Y}$), 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 $\widehat{L}_k$. 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 $\widehat{L}_k$, measure the difference, and do the model update to L_k, i.e.,

$$\mathcal{L}_{LKD,k} = MSE\left(L_k \cdot L_{k-1} \cdot L_{k-2} \cdot ... \cdot L_1(\boldsymbol{X}) - \widehat{L}_k \cdot L_{k-1} \cdot L_{k-2} \cdot ... \cdot L_1(\boldsymbol{X})\right), \tag{1}$$

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.

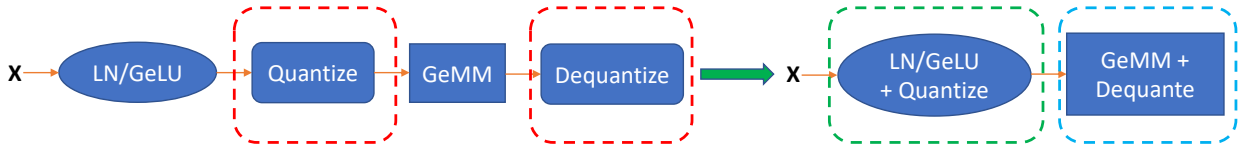


Figure 2: The illustration of normal (left) and our fused (right) INT8 GeMM.

Table 2: Result of $\text{BERT}_{\text{base}}$ on the development set of GLUE benchmark (except WNLI). [56]$^+$ uses 128 groups for weight matrix which is hard to get GPU acceleration. [6]* 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)$^+$	—	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 $\text{BERT}_{\text{base}}$ and $\text{BERT}_{\text{large}}$ on GLUE benchmark; and for GPT-3-style models, we tested the $\text{GPT-3}_{\text{350M}}$ (i.e., GPT-3-style model with 350M parameters) and $\text{GPT-3}_{\text{1.3B}}$ (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_{6B} [66] and $\text{GPT-NeoX}_{\text{20B}}$ [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 $\boldsymbol{W}_{q/k/v}$) 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

$\text{BERT}_{\text{base}}$ We report the results of $\text{BERT}_{\text{base}}$ 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.

$\text{BERT}_{\text{large}}$ We test our methods on $\text{BERT}_{\text{large}}$ as well and the results are shown in Table 3. Similar to $\text{BERT}_{\text{base}}$, 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 $\text{BERT}_{\text{large}}$, which is 13x cheaper than QAT.

5.2 Main Results of GPT-3-style Models

$\text{GPT-3}_{\text{350M}}$ We first test ZeroQuant and ZeroQuant-LKD on $\text{GPT-3}_{\text{350M}}$ 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 $\text{BERT}_{\text{large}}$ on the development set of GLUE benchmark (except WNLI). $^{+}$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)$^{+}$	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 $\text{GPT-3}_{\text{350M}}$ on 20 zero-shot evaluation datasets. Please see Table H.1 for the results of all 20 tasks.

Precision (Method)	Lambada (↑)	PIQA (↑)	OpenBookQA (↑)	RTE (↑)	ReCoRd (↑)	Ave. 19 Tasks (↑)	Wikitext-2 (↓)	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.

$\text{GPT-3}_{\text{1.3B}}$ The results of $\text{GPT-3}_{\text{1.3B}}$ are shown in Table 5. Similar to $\text{GPT-3}_{\text{350M}}$, 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 $\text{BERT}_{\text{base}}$ and 2.47–5.01x on $\text{BERT}_{\text{large}}$.

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 $\text{GPT-3}_{\text{350M}}$/$\text{GPT-3}_{\text{1.3B}}$ as compared to the FP16 counterpart.

Table 5: Post training quantization result of GPT-$3_{1.3B}$ on 20 zero-shot evaluation datasets. Please see Table H.2 for the results of all 20 tasks.

Precision (Method)	Lambada (↑)	PIQA (↑)	OpenBookQA (↑)	RTE (↑)	ReCoRd (↑)	Ave. 19 Tasks (↑)	Wikitext-2 (↓)	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	Precision	128								256							
BS		1	2	4	8	16	16	64	128	1	2	4	8	16	16	64	128
$BERT_{base}$	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
$BERT_{large}$	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_{6B} and GPT-$NeoX_{20B}$

To demonstrate the scalability of ZeroQuant, we applied it to two of the largest open-sourced models, i.e., GPT-J_{6B} and GPT-$NeoX_{20B}$, which have 6B and 20B parameters separately.

We report the results of GPT-J_{6B} 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-$NeoX_{20B}$ 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 $BERT_{large}$ 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_{6B} 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-$NeoX_{20B}$ 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 $BERT_{large}$ 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 $GPT\text{-}3_{350M}$ 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 $GPT\text{-}3_{350M}$ 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[1]), 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

[1] 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$_{20B}$). 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$_{20B}$.

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. Copa: 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, Shivanshu 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, 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.

[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, Mitsuhisa 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, Pierric 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 (MHSA) and the feed-forward connection (FFC). 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.

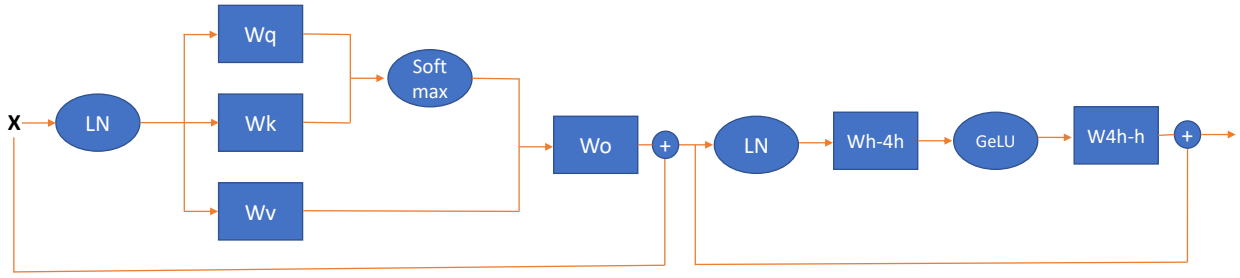


Figure A.1: The illustration of a Transformer-block.

Transformer Block Assume the input of an encoder layer is $\boldsymbol{X}$, the query, key, value, attention output, FFC dense, and FFC output matrices are $\boldsymbol{W}_q$, $\boldsymbol{W}_k$, $\boldsymbol{W}_v$, $\boldsymbol{W}_o$, $\boldsymbol{W}_{h-4h}$, and $\boldsymbol{W}_{4h-h}$, 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

$$\mathbf{x}_{quantize} = round\left(clamp(\frac{\mathbf{x}}{S}, -2^{bit-1}, 2^{bit-1}-1)\right), \tag{2}$$

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 $S = max\left(abs(\mathbf{x})\right)$, 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.,

$$x_{max} = 0.95x_{max} + 0.05max(x_{current-iteration}),$$
$$x_{min} = 0.95x_{min} + 0.05min(x_{current-iteration}).$$

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 $\text{BERT}_{\text{base}}$ and $\text{BERT}_{\text{large}}$. We use the same lower-case tokenizer in $\text{BERT}_{\text{large}}$ 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]).[2]), we follow the instruction from Huggingface Transformer Library [72].

For ZeroQuant and ZeroQuant-LKD, we use 48 groups for group-wise weight quantization on $\text{BERT}_{\text{base}}$ and 64 groups for group-wise weight quantization on $\text{BERT}_{\text{large}}$, for all the weight matrices.

For LKD, we use 100 iterations with batch size 32 and sequence length 128 for $\text{BERT}_{\text{base}}$, and we use 400 iterations for $\text{BERT}_{\text{large}}$. 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 [3]. 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 $\text{GPT-3}_{\text{350M}}$ and 120 hours of $\text{GPT-3}_{\text{1.3B}}$. 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 $\text{GPT-3}_{\text{350M}}$/$\text{GPT-3}_{\text{1.3B}}$ for all the weight matrices.

For LKD, we use 1600 iterations with batch size 8 and sequence length 2048 for both $\text{GPT-3}_{\text{350M}}$ and $\text{GPT-3}_{\text{1.3B}}$. 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.

[2] We exclude WNLI [35] since its results are not stable [16].

[3] 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 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	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 MHSA (i.e., the row of W4/8A16). This ultra-low precision quantization leads the model to be purely random without meaning prediction.

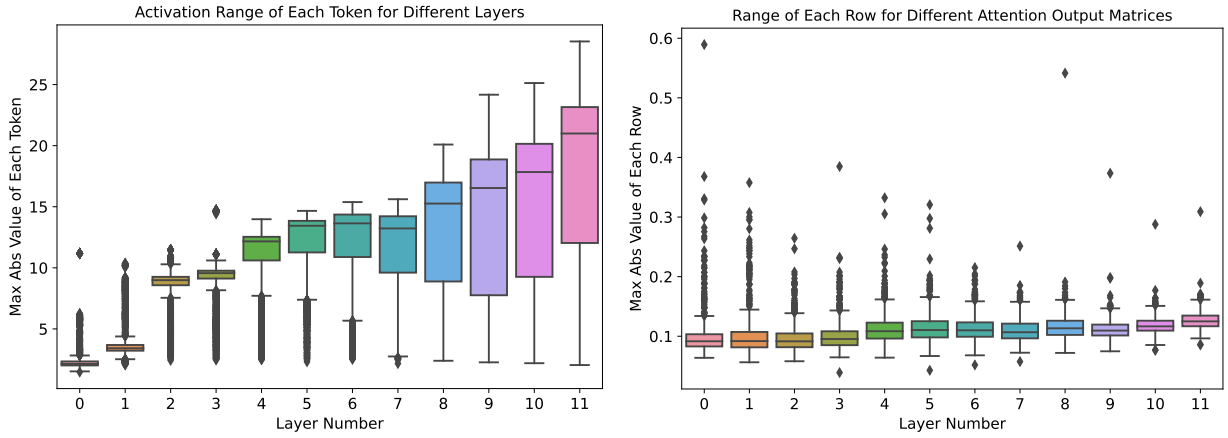


Figure C.1: The activation range of different layers (left) and the row-wise weight range of the attention output matrix ($\boldsymbol{W}_o$) of different layers (right). The results are based on the $\text{BERT}_{\text{base}}$ trained on MNLI dataset. Please see Figure 1 for the results of $\text{GPT-3}_{\text{350M}}$.

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 ($batch \times seq_len$), 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, WARPs, 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 $\text{BERT}_{\text{base}}$. 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 4×7181s

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 ($\text{GPT-NeoX}_{\text{20B}}$). 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_{350M}.

Tasks	Baseline	PTQ				ZeroQuant			ZeroQuant-LKD	
	W32A32	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, inlcuding GPT-$NeoX_{20B}$.

Table H.2: The full results of GPT-$3_{1.3B}$.

Tasks	Baseline	PTQ		ZeroQuant			ZeroQuant-LKD	
	W32A32	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_{350M} 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$_{20B}$.

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