COMET: Towards Practical W4A4KV4 LLMs Serving

Lian Liu
Institute of Computing Technology, CAS, University of Chinese Academy of Sciences
Beijing, China
liulian21s@ict.ac.cn

Long Cheng
North China Electric Power University
Beijing, China
lcheng@ncepu.edu.cn

Haimeng Ren
ShanghaiTech University
Shanghai, China
renhm2022@shanghaitech.edu.cn

Zhaohui Xu
ShanghaiTech University
Shanghai, China
xuzhh12022@shanghaitech.edu.cn

Yudong Pan
Institute of Computing Technology, CAS, University of Chinese Academy of Sciences
Beijing, China
panyudong23s@ict.ac.cn

Mengdi Wang
Institute of Computing Technology, CAS
Beijing, China
wangmengdi@ict.ac.cn

Xiaowei Li
Institute of Computing Technology, CAS, Zhongguancun Laboratory
Beijing, China
lxw@ict.ac.cn

Yinhe Han
Institute of Computing Technology, CAS
Beijing, China
yinhes@ict.ac.cn

Ying Wang*
Institute of Computing Technology, CAS
Beijing, China
wangying2009@ict.ac.cn

Abstract

Quantization is a widely-used compression technology to reduce the overhead of serving large language models (LLMs) on terminal devices and in cloud data centers. However, prevalent quantization methods, such as 8-bit weight-activation or 4-bit weight-only quantization, achieve limited performance improvements due to poor support for low-precision (e.g., 4-bit) activation. This work, for the first time, realizes practical W4A4KV4 serving for LLMs, fully utilizing the INT4 tensor cores on modern GPUs and reducing the memory bottleneck caused by the KV cache. Specifically, we propose a novel fine-grained mixed-precision quantization algorithm (FMPQ) that compresses most activations into 4-bit with negligible accuracy loss. To support mixed-precision matrix multiplication for W4A4 and W4A8, we develop a highly optimized W4Ax kernel. Our approach introduces a novel mixed-precision data layout to facilitate access and fast dequantization for activation and weight tensors, utilizing the GPU's software pipeline to hide the overhead of data loading and conversion. Additionally, we propose fine-grained streaming multiprocessor (SM) scheduling to achieve load balance across different SMs. We integrate the optimized W4Ax kernel into our inference framework, COMET, and provide efficient management to support popular LLMs such as LLaMA-3-70B. Extensive evaluations demonstrate that, when running LLaMA family models on a single A100-80G-SMX4, COMET achieves a kernel-level speedup of 2.88× over cuBLAS and a 2.02× throughput improvement compared to TensorRT-LLM from an end-to-end framework perspective.

CCS Concepts: • **Computer systems organization → Parallel architectures**; • **Computing methodologies → Machine learning**.

Keywords: Large Language Models (LLM) Serving, LLM Quantization, Algorithm-System Co-design

ACM Reference Format:
Lian Liu, Long Cheng, Haimeng Ren, Zhaohui Xu, Yudong Pan, Mengdi Wang, Xiaowei Li, Yinhe Han, and Ying Wang. 2025. COMET: Towards Practical W4A4KV4 LLMs Serving . In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676641.3716252

*Corresponding author.

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/25/03.
https://doi.org/10.1145/3676641.3716252

1 Introduction

Large language models (LLMs) have demonstrated excellent performance across various benchmarks [6, 10, 41, 50, 55, 62]. However, as LLMs advance and models with hundreds of billions of parameters emerge, their substantial sizes present significant challenges for inference systems. Specifically, large models require extensive memory, while most LLM-based systems perform inference on a single GPU with limited

memory capacity. Moreover, LLM inference incurs high serving costs, often calculated per token, and processing long token sequences further increases these costs.

Model quantization is an efficient way to reduce the memory footprint and serving costs for LLM inference, with weight-only quantization being a typical method in recent years [12, 28]. However, the latest studies [29, 65] report that weight-only quantization achieves limited performance improvements on modern GPUs, particularly when processing large-batch and long token sequences. The main reasons are: (1) weight-only quantization requires low-bit parameters to be dequantized to align with high-precision activations before being processed by the GPU tensor cores, leading to a waste of computational resources. For example, existing W4A16 quantization methods [12, 28, 48] must restore the quantized 4-bit weights to 16-bit and process them together with activations in the FP16 tensor cores, which is inefficient for modern GPUs like A100 that are optimized for higher-throughput 4-bit operations; and (2) in applications involving large-batch processing and long token sequences [18, 60], the Key and Value activation caching (KV cache) becomes the major bottleneck rather than the weight parameters. Although methods like SmoothQuant [56] simultaneously quantize both activations and weights, they employ a conservative scaling strategy that restricts activations and weights to INT8 format, still facing the issues mentioned above.

To address the above issues, achieving lower-precision (e.g., 4-bit) activations without compromising accuracy is crucial. This would fully utilize low-bit tensor cores in modern GPUs, delivering higher throughput. Additionally, low-precision quantization for the KV cache, which consumes significant memory in transformers, is needed. This would alleviate the memory bottleneck, enable larger inference batch sizes, and efficiently exploit the batch-level parallelism in advanced GPUs. These requirements, along with the concentrated distribution of outliers in activations [10, 49], motivate us to design a novel fine-grained mixed-precision quantization algorithm (FMPQ) for activations. Specifically, FMPQ quantizes most activations to 4-bit and others to 8-bit[1]. To ensure efficient computing, we partition the activation tensor into multiple sub-tensors, each sized to match the GPU's computational granularity. Additionally, we introduce a channel permutation strategy to cluster outliers within the same sub-tensor, thereby reducing the overall quantization precision.

Since FMPQ requires hardware capable of W4Ax matrix multiplication for the quantized LLM, but existing LLM serving systems [14, 40, 46] lack support for direct mixed-precision tensor operations and W4Ax computing, we further design a novel W4Ax kernel and integrated it into our inference framework, COMET. Generally, COMET optimizes mixed-precision LLM computing on GPUs by incorporating data layout design and fast dequantization for mixed-precision weights and activations. It further uses the software pipeline to overlap the overhead of data loading and conversion. Additionally, given that the lower precision tensor cores in modern GPU provide higher throughput (INT4 tensor core has 2× higher throughput than INT8), COMET employs a fine-grained SM scheduling strategy to achieve load balance across different stream multiprocessors (SMs). By integrating the optimized W4Ax kernel and efficient memory management techniques [23], COMET provides practical and efficient W4A4KV4 LLM serving.

In a nutshell, the contributions of this work can be summarized as follows:

- We analyze the distribution of outlier values in LLM activations and introduce a novel FMPQ algorithm that enables 4-bit activations and KV cache without compromising accuracy. To achieve this, the activation tensor is divided and quantized at a granularity that matches the matrix multiplication units on modern GPUs, employing a tiling approach. With negligible accuracy loss, the proposed FMPQ algorithm processes more than 84% of GEMM computations using W4A4, while W4A8 is used for the remaining computations.
- We develop a novel highly-optimized W4Ax kernel to support the simultaneous computation of W4A4 and W4A8. The low-precision data points are packed into a high-precision format and directly processed in CUDA cores using an optimized pipeline, effectively hiding the expensive runtime numerical-format conversion overhead. Furthermore, we propose an efficient fine-grained SM scheduling solution during LLM compilation stages. This solution remaps the mixed-precision tensor tiles, to achieve balanced mixed-precision computing across different SMs.
- We present COMET, the high-performance LLM inference framework, which integrates our proposed W4Ax kernel to enable mixed-precision GEMM processing and provides efficient memory management for LLM serving. Compared with state-of-the-art (SOTA) frameworks, COMET achieves practical W4A4KV4 LLM serving. Evaluated on a single A100-80G-SXM4 across various LLM models, COMET demonstrates a 1.48 × − 2.91× latency reduction on kernel performance and 2.02× throughput improvement in end-to-end evaluation over SOTA baselines. Additionally, we provide an open-source W4Ax kernel[2] with a Python interface and a set of C++ APIs, enabling seamless integration into existing inference systems such as TensorRT-LLM [40] and DeepSpeed [46].

[1]Throughout this paper, *mixed-precision* refers to a combination of W4A4 and W4A8 (i.e., **W4Ax**), rather than mixing 8-bit or 16-bit activations with lower-precision weights as seen in current works [28, 29].

[2]Open-sourced at https://github.com/rhmaaa/COMET-LLM

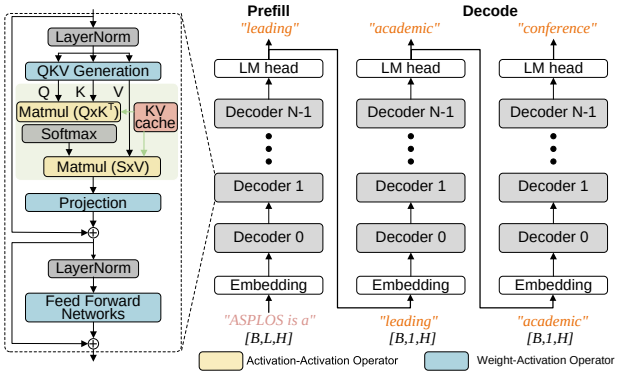


Figure 1. The inference procedure of LLMs is divided into two phases: prefill and decode.

2 Background & Motivation

2.1 LLM Inference

Figure 1 illustrates the inference procedure of LLMs, structured into two distinct phases: prefill and decode. In the prefill phase, the model receives an input sentence and processes multiple tokens simultaneously, allowing for parallel computation. The input tensor shape during this phase is $[B, L, H]$, where B is the batch size, L is the prompt sequence length, and H represents the hidden dimension of the model. In the decode phase, by contrast, the model generates the output sequence in an auto-regressive manner, producing one token at a time. This sequential process adjusts the input tensor shape to $[B, 1, H]$, as only a single token is processed at each step.

It is important to note that during token decode, the model needs to store intermediate Key and Value activations to avoid redundant computations. This storage is commonly referred to as the KV cache. During inference, the system must preserve not only all weight parameters but also the Key-Value pairs generated during the decode phase. While the storage space for weights remains constant, the size of the KV cache is directly proportional to the sequence length. Consequently, as sequence length increases, the KV cache gradually becomes the primary storage bottleneck, overtaking weight parameters. For instance, at a sequence length of 128K, the KV cache accounts for 72% of the total storage cost for LLaMA-7B [18].

2.2 LLM Quantization

Quantization is an effective method for reducing the inference cost of LLMs. Existing quantization methods for LLMs fall into two categories: weight-only and weight-activation quantization [60, 64].

Weight-only Quantization. Weights are more amenable to quantization compared to activations [4, 12, 28]. Consequently, a substantial body of research has concentrated on achieving low-precision weights. Given the high retraining costs associated with LLMs, existing studies mainly focus on post-training quantization (PTQ). Methods such as GPTQ [12], QuIP [7], and QuIP# [53] achieve low-precision weights by minimizing layer-wise quantization error and optimizing parameters. Additionally, AWQ [28] and OWQ [24] consider the impact of activation outliers on weight quantization, resulting in improved performance for LLMs. OmniQuant [48] further introduces learned weight clipping parameters to achieve lossless W4A16 quantization. While weight-only quantization can reduce the storage costs for modern inference systems, it has limited ability to decrease computational costs and provides minimal performance improvement for processing long sequences. Therefore, our work primarily focuses on achieving low-precision activation quantization.

Weight-activation Quantization. Unlike weight-only quantization methods, weight-activation quantization techniques target both weights and activations, including the KV cache. As discussed in [10], the biggest challenge in quantizing activations is handling outliers, which can be orders of magnitude larger than typical values. To address this, SmoothQuant [56] introduces an equivalent transformation strategy that partially shifts the quantization difficulty from activations to weights, achieving a practical solution of W8A8. To further enhance the accuracy of quantized models, some works have explored fine-grained quantization strategies specifically tailored for activations. For example, KVQuant and WKVQuant [18, 60] focus on achieving fine-grained quantization for the KV cache by adopting per-token or per-channel quantization strategies. Although these approaches can reduce memory usage, they are inefficient for computing. ZeroQuant [58] and Atom [65] apply per-channel quantization for activations, while RPTQ [59] proposes a reordering strategy to aggregate non-uniformly distributed channels into different activation parts. Moreover, recent works [4, 32] further employ group-wise quantization to improve the accuracy of quantized LLMs. However, these aggressive quantization strategies introduce significant dequantization overhead, which cannot be efficiently processed in modern GPUs [29].

Overall, existing weight-only and weight-activation quantization methods for LLMs focus heavily on quantization accuracy, often neglecting system-level efficiency, which results in suboptimal inference performance. Specifically, these methods cannot fully unleash the potential computation capability of tensor cores within modern GPUs due to a misalignment between the quantization granularity and the computational granularity of GPUs. In comparison, we propose a fine-grained quantization strategy that aligns with the computational granularity of GPUs. Additionally, we utilize mixed-precision techniques to further improve the compression ratio of the LLM, thereby enhancing both memory efficiency and computational performance.

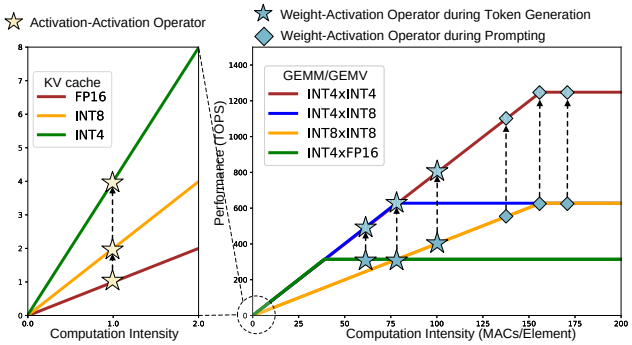
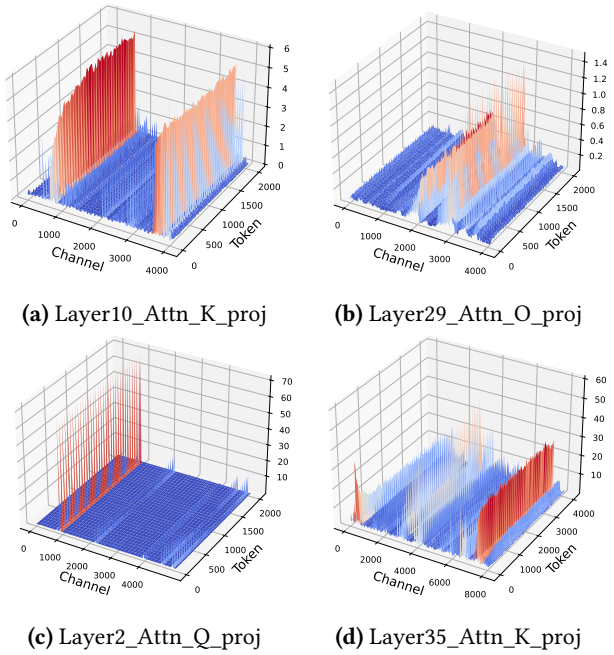


Figure 2. Roofline model analysis for activation-activation and weight-activation operators. The activation-activation operator is highly memory-bound, while the weight-activation operator becomes compute-bound with large-batch parallelism.



(a) Layer10_Attn_K_proj **(b)** Layer29_Attn_O_proj

(c) Layer2_Attn_Q_proj **(d)** Layer35_Attn_K_proj

Figure 3. Examples of activation distributions for various LLMs: (a) and (b) show LLaMA-7B activations, (c) presents OPT-13B activations, and (d) illustrates Qwen2-72B activations. Certain activation channels contain outliers, which can degrade accuracy when applying conventional quantization strategies.

2.3 Motivation

Existing LLM serving systems [40, 46] typically utilize modern GPUs, which are equipped with high-bandwidth memory (HBM) and powerful processing units such as high-throughput tensor cores. For instance, the A100 80G offers 80GB of HBM with 2.0TB/s bandwidth and can deliver up to 312 TFLOPS of FP16 tensor core performance for GEMM operations. In addition, modern GPUs support lower-precision computations, achieving 624 TOPS with INT8 tensor cores and 1248 TOPS with INT4 tensor cores. Modern GPUs also include CUDA cores for handling complex computations such as data conversion and permutation.

As depicted in Figure 2, we employ the classic roofline model to evaluate the performance of LLM inference on GPUs. According to Figure 1, we assess the performance of two types of operators, activation-activation operators and weight-activation operators, under different precisions. The computation intensity of the activation-activation operator is fixed at 1.0, making it memory-bandwidth bound. Therefore, implementing low-bit quantization for the KV cache can significantly enhance the throughput of activation-activation operators. Conversely, the computation intensity of weight-activation operators varies with batch size and inference phases. Thus, under large-batch parallelism, adopting low-precision activation quantization can further improve inference throughput. These observations motivate us to explore low-precision quantization for activations (including input activations and the KV cache) more thoroughly. Additionally, due to being bound by different characteristics, we will adopt different quantization strategies for input activation and KV cache, as detailed in Section 3.2.

3 FMPQ: Fine-Grained Mixed-Precision Quantization

To address the challenges of activation quantization in post-training quantization, this section first analyzes the characteristics of activation distributions in LLMs and then proposes a mixed-precision quantization algorithm to achieve low-bit quantization of LLM activations. The proposed FMPQ effectively reduces the computational and storage costs of LLM inference, serving as the foundational enabler of our COMET inference framework.

3.1 Analysis of Activation Distribution

Unlike small-scale neural networks [11, 31, 54], a notable feature of LLMs is the widespread occurrence of outliers.Once a model exceeds a certain scale (in practice, around 6 billion parameters), a small cluster of hidden state features (usually less than 1%) exhibits magnitudes significantly larger than the rest [10]. These emergent large-magnitude features are referred to as outliers, and they have magnitudes that can exceed typical hidden state values by tenfold or even a hundredfold. When using traditional min-max quantization strategies on activation tensors, these outliers can cause typical hidden state values to be quantized to zero, resulting in significant accuracy degradation. These outliers are so

crucial to the representational capacity of LLMs that their removal is not feasible [15].

Fortunately, we have conducted extensive experimental analysis on current typical LLMs, such as LLaMA, and found that these outliers typically distribute within specific feature dimensions. As illustrated in Figure 3, outliers appear only in certain activation channels across various LLMs. This insight suggests an intuitive approach: applying separate quantization for outliers and normal values, with high precision for outliers and lower precision for normal values.

3.2 The FMPQ Algorithm

With the above observation, we can apply a fine-grained mixed-precision quantization algorithm that partitions the normal values and outliers into different parts and quantizes them with different precision. For example, we quantize normal values to 4-bit, while outliers to 8-bit. However, as demonstrated in Figure 4(b), previous channel-wise quantization algorithms [58, 65] cannot ensure the efficiency of the quantized model due to their misalignment with the granularity of the hardware computation.

In fact, the minimum computation granularity on modern GPUs is fixed. For instance, the FP16 tensor core on A100 has a minimum computation granularity of $64\times64\times32$ ($m\times n\times k$). This hardware constraint necessitates that the granularity of quantization is an integer multiple of the minimum computation granularity, to maximize the utilization of tensor core. Consequently, prior channel-wise or irregular cluster quantization strategies [58, 59], which fail to align well with hardware computation granularity, are inefficient. On the other hand, overly fine-grained quantization strategies (e.g., tile-wise quantization) introduce significant software scheduling overhead. Therefore, as illustrated in Figure 4(c), we adopt a block-wise mixed-precision quantization method, which partitions the activation tensor only along the channel dimension. Specifically, we divide the activation tensor into multiple sub-tensors with a channel size of k, referring to these sub-tensors as blocks. In practice, setting k to 128 ensures sufficient tensor core utilization without excessive accuracy loss in the quantized LLM.

Since outliers are distributed across multiple channels, any block that contains outliers will require 8-bit quantization. This results in a large number of blocks being quantized to 8-bit, which limits the benefits, as depicted in Figure 4(c). To address this issue, we employ a channel permutation strategy, which is commonly used in LLM pruning [44, 63, 65]. Unlike approaches that use channel permutation to achieve accurate N:M sparsity by distinct channel aggregation, we focus on identifying and clustering channels with outliers to minimize high-precision quantization. As shown in Figure 4(d), we utilize sampled activations from the calibration dataset [45] to locate outlier channels and apply permutation to group these channels into a single block. To further maintain computational equivalence, the corresponding positions

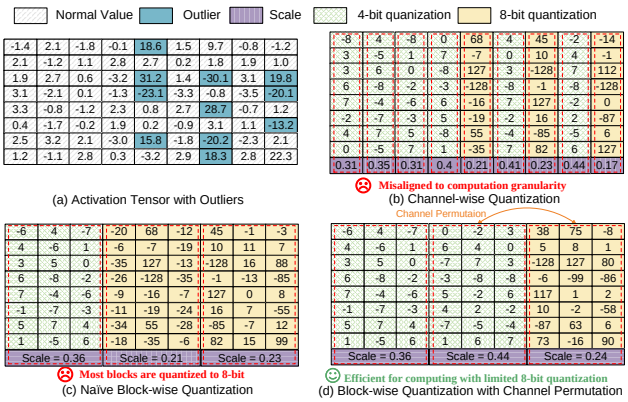


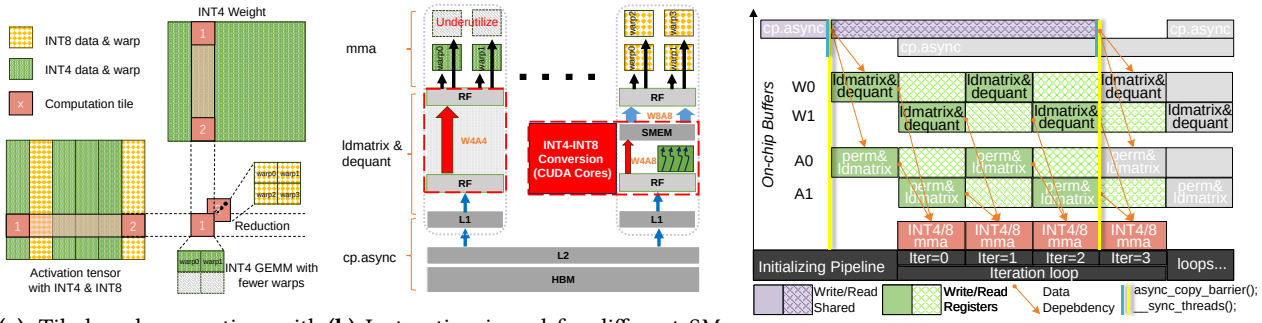
Figure 4. The algorithm design of FMPQ.

in the weight matrix also need to be permuted. In this way, only a small portion of blocks (less than 20%) need 8-bit quantization, while the majority of activations can be quantized to 4-bit, enabling W4A4 GEMM. Prior works [27, 59] have already optimized the channel permutation operator, and according to our evaluation, channel permutation accounts for only 0.7% of the overall runtime.

While the proposed channel permutation enhanced block-wise quantization works well for input activations, the KV cache requires a different quantization strategy. As analyzed in Figure 2, the activation-activation operator which KV cache works for is highly memory-bound. Therefore, KV cache is better suited for low-bit quantization without considering the quantized granularity. Given that RoPE and softmax possess strong outlier regularization properties, low-precision quantization of the K cache introduces minimal error. Furthermore, the V cache contains few outliers [18, 34, 65]. As a result, we can apply a full 4-bit quantization to the KV cache. Specifically, we use a channel-wise 4-bit quantization strategy for the KV cache. The experimental results in Section 6.2 demonstrate that this approach has a negligible impact on accuracy. By combining the quantization for input activation and KV cache, the FMPQ algorithm provides a practical path for W4A4KV4 LLM serving.

4 COMET-W4Ax: Kernel Design

The proposed FMPQ algorithm can significantly reduce the storage and computing costs in LLM inference. However, existing LLM serving systems [14, 40] lack support for direct mixed-precision tensor load-and-store and W4Ax computing. Thus, in this section, we design a highly optimized W4Ax kernel for COMET by tackling two main challenges: (1) the additional overhead of data management with mixed-precision encoding, and (2) load imbalance induced by varied W4A4 and W4A8 GEMM operations.



(a) Tile-based computing with mixed-precision.

(b) Instruction issued for different SMs with different precision.

(c) SIMT-enhanced software pipeline.

Figure 5. The design overview of the COMET-W4Ax kernel. (a) illustrates the tile-based GEMM computing with mixed-precision encoding. (b) presents the computing procedure when issuing W4A4 and W4A8 GEMM instructions simultaneously. (c) shows the two-level overlapping within SIMT-based software pipeline

4.1 Design Overview

GEMM computations on GPUs are performed at the tile granularity. Figure 5(a) illustrates the tile-based GEMM computing in COMET using mixed-precision values. With FMPQ as the algorithmic enabler, the activation tensor is divided into multiple blocks with different precision settings. For example, the green block is quantized to 4-bit, while the yellow block is 8-bit. A block usually contains multiple tiles, and each tile invokes one thread block (TB) to compute. Additionally, we utilize the reduction operator to accumulate the compute results of different tiles across multiple TBs.

Figure 5(b) shows the behavior of the COMET-W4Ax kernel when computing different precision tiles among different SMs at the same time. During the kernel execution, data must first be loaded from global memory into shared memory and then dispatched to each SM's tensor core for computation. As presented in Section 4.2, we use a software pipeline to concurrently handle data loading and GEMM computation. COMET-W4Ax primarily involves two types of GEMM computations during kernel execution: W4A4 and W4A8. The W4A4 computation can directly leverage the `mma` instructions, whereas the W4A8 GEMM requires additional data conversion. Specifically, we utilize CUDA cores to efficiently convert INT4 to INT8 data formats and store the converted results directly in shared memory to support W4A8 GEMM. We present the detailed optimization on data conversion in Section 4.3. Additionally, we notice that when issuing GEMM computation instructions for W4A4 and W4A8 to different SMs simultaneously, the computational resources of the tensor cores used for W4A4 are not fully utilized. Despite the INT4 tensor cores offering 2× higher throughput than INT8 tensor cores, the SMs executing W4A4 computations have to wait for other SMs to achieve synchronization, leading to significant idle times and low resource utilization. In Section 4.4, we address this issue using fine-grained SM scheduling.

4.2 SIMT-enhanced Software Pipeline

Although load-compute pipelining is a common optimization to overlap data loading and computation on GPUs [19, 22], achieving an efficient pipeline for the W4Ax kernel is challenging. Unlike traditional pipelines that rely on tensor cores for data loading and GEMM operations, W4Ax requires additional steps such as dequantization and permutation, which must be handled by CUDA cores. With CUDA cores having much lower throughput than tensor cores (e.g., 78 TFLOPS vs. 1248 TOPS on the A100), these extra steps cause bottlenecks if not effectively overlapped. To address this, we propose a novel SIMT-enhanced software pipeline that mitigates the overhead of these data transformation operations. Our approach integrates multiple processing stages that seamlessly handle these additional transformations, enabling efficient overlap and minimizing performance degradation.

As demonstrated in Figure 5(c), COMET employs a two-level overlapping strategy to pipeline the memory access and computation stages, enhancing GEMM computation efficiency. First, it hides off-chip memory loads within the data transformation and tensor core computation phases. Second, it employs double buffering within the GPU to conceal the overhead of tensor core computation and data transfer/transformation. Specifically, COMET uses two shared memory buffers to store different data tiles (storing A_0 and W_0 in $buffer_0$, and A_1 and W_1 in $buffer_1$). In iteration 0, the tensor core loads A_0 and W_0 from $buffer_0$ and performs the corresponding GEMM computation. Depending on the data format loaded, the tensor core invokes different `mma` instructions (INT4 or INT8). Simultaneously, $buffer_1$ loads the relevant data for A_1 and W_1 from global memory and decides whether permutation or dequantization adjustments are necessary. In iteration 1, the tensor core loads data from $buffer_1$ while $buffer_0$ prefetches data from global memory. In this way, data loading and computation are effectively overlapped across both CUDA and tensor cores.

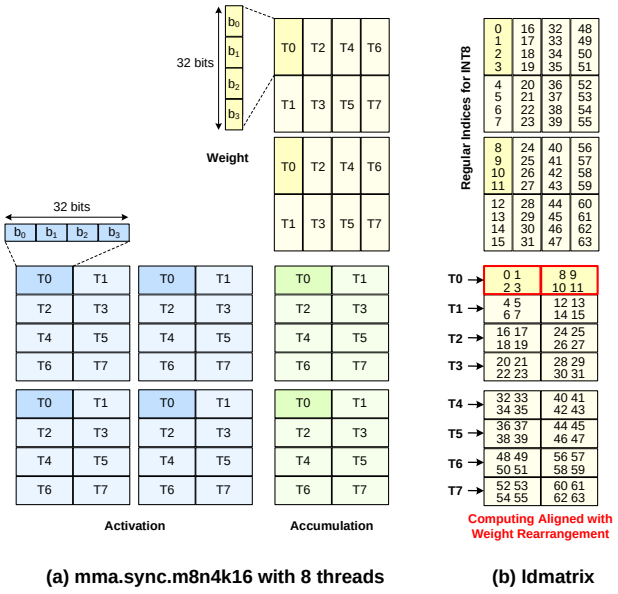


Figure 6. Weight interleave for fast access of W4A8 GEMM. (a) Typical W8A8 GEMM computing kernel. (b) Optimizing the `ldmatrix` instruction when loading INT4 weight to INT8 GEMM kernel.

Given the complexity of the above pipeline, thread synchronizations and memory access barriers are crucial to ensure correct kernel execution. COMET specifically uses `async_copy_barrier` to guarantee that all necessary data is loaded into shared memory before the next iteration begins, while `sync_threads` ensures full thread synchronization at the start of each iteration.

4.3 Mixed-precision Data Management

The COMET-W4Ax kernel involves GEMM computations of different precisions, including W4A4 and W4A8. However, the GPU tensor core only supports GEMM computations with data in the same format. Therefore, to optimize the efficiency of W4A8 computations, we need to implement efficient data layout and data conversion strategies.

Weight Interleaving for W4A8 GEMM. Before performing actual computations on the GPU, all operands must be loaded from shared memory into the corresponding registers for each tensor core (`ldmatrix`). Subsequently, a thread scheduler is used to load the required data into the tensor core for the corresponding GEMM computation (`mma`). For W4A8 GEMM, the tensor core performs computations using INT8 tensor cores to execute W8A8 computations. The tensor core relies on the `mma.m16n8k32` instruction to perform INT8 GEMM computations, involving 32 threads. Given the complexity of the `mma` execution, Figure 6(a) shows a simplified example of `mma.m8n4k16` GEMM computation which only uses 8 threads for INT8 GEMM. As it illustrates, each

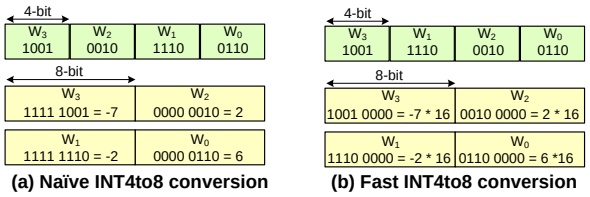


Figure 7. Fast data conversion from INT4 to INT8.

thread needs to load activation data four times and weight data two times, then conduct the multiply-accumulate (MAC) operations two times for a naïve W8A8 GEMM.

In existing GEMM optimization frameworks, as long as the format of the loaded data matches the computation data format, programmers can directly use the `ldmatrix` instruction to ensure the loaded data corresponds to the required computation data. However, directly transferring 4-bit weights stored in an 8-bit format to the tensor core can cause shared memory conflicts. These conflicts occur when multiple threads within the same streaming multiprocessor (SM) simultaneously attempt to access the same shared memory address. As shown in Figure 6(a), in a typical W8A8 GEMM kernel, thread T0 loads the first 32 bits, including four values $b_0 \sim b_3$, while thread T1 loads the next four values $b_4 \sim b_7$. However, as weights are encoded in INT4 format in the W4A8 GEMM case, thread T0 needs to load $b_0 \sim b_7$ while T1 loads $b_4 \sim b_{11}$, resulting in overlap to the same set of values $b_4 \sim b_7$. This overlap introduces shared memory conflicts, making data loading both costly and inefficient.

To prevent shared memory conflicts, we rearrange the weight data layout. As shown in Figure 6(b), each thread now loads non-contiguous weights. Specifically, thread T0 uses addresses $0 \sim 3$ and $8 \sim 11$ instead of the contiguous range $0 \sim 7$. This interleaving strategy not only eliminates shared memory conflicts but also reduces the need for `ldmatrix` instructions, achieving the required data load with a single instruction instead of the two needed for the INT8 scenario.

Fast INT4to8 Conversion. Since tensor cores only support GEMM computations with data in the same format, it is necessary to convert INT4 data to INT8 before performing GEMM computation. Data format conversion can only be executed on CUDA cores. However, in modern GPUs, there is a significant performance gap between CUDA cores and tensor cores. For example, on the A100, the computational throughput of the INT8 tensor core is 32× higher than that of the CUDA core. Therefore, we must implement a fast INT4 to INT8 data format conversion, making it a stepping stone for performance enhancement rather than a bottleneck.

Suppose we pack four 4-bit weight values into a single 16-bit data unit and store them sequentially. Figure 7a illustrates a naïve strategy for converting INT4 to INT8. In this process, data position adjustments and sign extensions are required. For example, in the conversion of W_1 and W_0, we first need

to shift the data position of W_1 from bits 4-7 to 8-11, and then perform sign extension on both W_0 and W_1. Unfortunately, the PTX ISA [39] does not support 4-bit shift operations or the corresponding sign extension operations, requiring us to rely on multiple instructions to complete the data conversion. Overall, each conversion can take up to 10 instructions to complete the necessary processing.

To mitigate the overhead of data position adjustments and sign extension, we propose two novel strategies for fast value conversion. First, we use a location switch to swap the storage positions of certain data. As shown in Figure 7b, we swap the positions of W_1 and W_2, enabling a rapid conversion to the corresponding 8-bit data format addresses. Additionally, we employ zero extension instead of sign extension for data conversion. Unlike INT4 sign extension which relies on multiple instructions to achieve, zero extension, on the other hand, can be achieved directly with a single zero-padding instruction. During zero extension, each value is effectively multiplied by 16, so we only need to divide by 16 in the scaling parameter to ensure consistency in the computation results. In comparison to naïve conversion, the overhead with our proposed strategy for each value is reduced to just 2 instructions for each conversion. This significantly improves the efficiency of data conversion.

To illustrate compatibility with the A100, we focus primarily on optimizing fast INT4-to-INT8 conversion here. It should be noted that the design is also adaptable for efficient FP4-to-INT8 conversion on next-generation GPUs such as H100. In the FP4 format [30], the sign and mantissa bits remain in their original positions, while the exponent bits are converted using typical shift instructions. For example, an exponent bit pattern of '01' corresponds to $2^{1-2} = 2^{-1}$, which translates to a signed right shift on the INT8 data.

4.4 Fine-Grained SM Scheduling

Given our implementation of fine-grained mixed-precision quantization, different blocks in the COMET system may require different precision `mma` instructions for computation. Figure 8(a) illustrates such a scenario on a hypothetical GPU with four streaming multiprocessor cores. Specifically, we set the block size $k = 128$. Consider a GEMM operation of size $256 \times 256 \times 384$, which includes two activation blocks in INT4 and INT8 formats, we divide the computation into 18 tiles. The shape of each tile is $128 \times 128 \times 128$. During the GEMM computation process, every two consecutive tiles compute the W4A8 and W4A4 GEMM kernels respectively.

Although W4A4 GEMM computations achieve higher throughput, they encounter stalls due to inserted barriers. As demonstrated in Figure 8(b), in each iteration, SM_1 and SM_3, which perform GEMM using the INT4 `mma` instruction, must wait for SM_0 and SM_2 to complete their INT8 `mma` computations to maintain synchronization. This requirement significantly reduces SM utilization. A straightforward solution to this under-utilization is to adopt the conventional heterogeneous task scheduling strategy [25, 26] to distribute different computing tiles globally. However, within the W4Ax kernel, different computing tiles require frequent reductions to maintain correctness, limiting the effectiveness of this scheduling approach. To this end, we propose an error-free barrier minimization technique and develop an efficient tile remapping strategy for fine-grained SM scheduling without compromising synchronization. Additionally, as indicated by the red dotted box, the imbalance between the number of tiles and SMs leaves some SMs idle in the final iteration, further decreasing overall efficiency. To resolve this, we introduce an innovative tile decomposition strategy that promotes load balancing across SMs within a single tile.

Synchronization Barrier Minimization. It is unnecessary to insert synchronization barriers for every `mma` iteration. We notice that synchronization is only necessary after all iterations are completed, as illustrated in Figure 8(c). Frequent synchronization barriers introduce additional communication overhead between SMs. Therefore, to minimize interaction between SMs, we should reduce the insertion of synchronization barriers as much as possible while ensuring computational correctness. The only synchronization required between SMs is the one that before writing the accumulation data back to memory. However, the performance gains from minimizing synchronization barriers are limited. Specifically, as one can notice, the SMs executing INT4 `mma` (SM_1 and SM_3) always wait for SM_0 and SM_2 to complete their corresponding computations before synchronization. As a result, the utilization of SM_1 and SM_3 remains low compared to SM_0 and SM_2.

Tile Remapping. Mapping all INT4 and INT8 `mma` computations to fixed SMs is unnecessary. To address this, we propose an effective tile remapping strategy that adjusts the mapping relationship between tiles and SMs to achieve effective load balancing across different SMs. As illustrated in Figure 8(d), we distribute the INT4 and INT8 `mma` computations as evenly as possible across all SMs. This ensures that each SM has a balanced computational load, reducing the overall kernel execution time. However, due to the mismatch between the number of divided tiles and the number of SMs, some SM resource wastage still occurs. Fine-grained tuning is required to further enhance the execution efficiency of mixed-precision GEMM.

Tile Decomposition. To achieve better alignment between tile division and SM resources, the most straightforward approach is to implement finer-grained tile division. However, this finer-grained blocking factor is less cache and scratchpad efficient, potentially negating any practical performance improvements. In this paper, we reexamine the binding relationship between tiles and SMs. The tile-to-SM binding is traditionally one-to-one, meaning that at any given time, a tile is mapped to a single SM. However, through effective scheduling, we can achieve a one-to-many

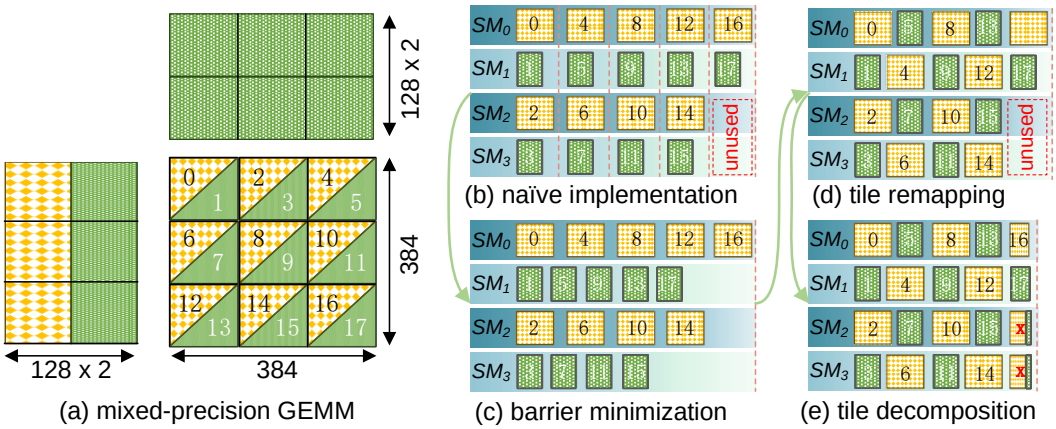


Figure 8. The proposed fine-grained SM scheduling. Instead of using a naïve implementation, we further adopt tile remapping and tile decomposition to greatly improve the tensor core utilization.

binding [42] between tiles and SMs. Specifically, we implement a task-stealing mechanism, where idle SMs can steal computational tasks from nearby busy SMs to balance the workload, as illustrated in Figure 8(e). For instance, when SM_0 and SM_1 are processing tile-16 and tile-17 respectively, SM_2 and SM_3 remain idle. During this period, SM_2 and SM_3 load data from shared memory and assist SM_0 and SM_1 with portions of their computations. This mechanism enhances the utilization of SMs and reduces the overall execution time.

5 COMET: System Implementation

We have implemented a practical W4A4KV4 serving framework, COMET, for high-performance LLM inference. COMET provides an easy-to-use Python interface for programmers. COMET leverages KV cache management optimizations from vLLM [23]. Furthermore, we introduce the W4Ax kernel to support mixed-precision matrix multiplication $O = WX$, where W is the weight matrix and A is the activation matrix. The COMET-W4Ax kernel has been implemented incorporating an additional 7,000 lines of C++ and CUDA code based on TensorRT-LLM [40] and CUTLASS [36]. The W4Ax kernel can be compiled as a standalone `.so` dynamic library. We also provide a set of C++ APIs so that programmers can easily integrate the optimized kernel in existing inference systems such as TensorRT-LLM and llama.cpp [14]. To further simplify usage, we use `pybind` to bind these interfaces to Python. This allows programmers to quickly and easily integrate our kernel into Python-based frameworks, such as Pytorch and Huggingface [20].

In our implementation, the tile size presented in Figure 5(a) is set as $128 \times 128 \times 128$ ($m \times n \times k$) for most cases. Furthermore, we set the warp shape as $64 \times 64 \times 128$ for INT4 tensor core, while $64 \times 64 \times 64$ for INT8 tensor core. As a result, the number of warps issued by one W4A4 tile is only half of the W4A8 tile. These configurations are selected based on extensive preliminary evaluation and are intended to optimize the computational throughput while balancing memory access patterns and processing power. By maintaining these fixed configurations across all our experiments, we aim to clearly demonstrate the performance improvement from detailed kernel optimization. The fixed configuration allows us to isolate the effects of other variables and focus on analyzing the impact of different optimization strategies and algorithmic adjustments. The chosen shapes reflect the typical dimensions used in high-performance computing scenarios and are well-suited to the architecture of modern GPUs, facilitating efficient parallel processing and minimizing latency.

6 Evaluation

6.1 Experimental Setup

Algorithm. The proposed quantization algorithm, FMPQ, is implemented using HuggingFace on top of PyTorch. We employ block-wise mixed-precision (INT4 and INT8) quantization for activations and channel-wise asymmetric INT4 group quantization for the KV cache. Additionally, we adopt the algorithm in [48] to achieve 4-bit weight quantization. We use the term "**W4AxKV4**" to denote the configurations we adopt. Note that most of the computations are conducted on W4A4KV4 cases.

System. We evaluate the performance of the COMET inference system at two different levels: kernel-level benchmarking and end-to-end inference performance. All performance evaluations are conducted on the NVIDIA A100-80GB-SXM4 platform with CUDA 12.1. Our primary focus is on the performance of linear layers within LLMs during the decode phase. GPU kernel performance is measured using NVIDIA Nsight Compute [37]. End-to-end inference throughput is measured using NVIDIA Nsight Systems [38]. We use TensorRT-LLM (TRT-LLM) v0.10.0 to perform inference

Table 1. Evaluation of WikiText2 perplexity for various quantized LLMs. Lower values indicate better performance.

		LLaMA-1			LLaMA-2			LLaMA-3		Mistral	OPT	Qwen2
Precision	Method	13B	30B	65B	7B	13B	70B	8B	70B	7B	13B	72B
FP16	-	5.09	4.10	3.56	5.12	4.57	3.12	6.14	2.86	5.25	10.13	4.94
W8A8	SmoothQuant	5.19	4.23	3.75	5.54	4.95	3.36	6.28	2.99	5.29	10.47	5.14
W4A16	GPTQ	5.40	4.48	3.83	5.83	5.13	3.58	7.02	3.44	5.39	10.31	5.09
	AWQ	5.34	4.39	3.76	6.15	5.12	3.54	7.09	3.40	5.37	10.39	5.07
	Omniquant	5.21	4.25	3.71	5.74	5.02	3.47	6.81	3.29	5.31	10.30	5.03
W4Ax	**FMPQ**	5.29	4.27	3.78	5.71	5.10	3.48	6.88	3.36	5.35	10.34	5.05
W4A4	Omniquant	10.87	10.33	9.17	14.26	12.30	9.93	14.27	9.75	7.87	11.65	7.91
W4A8 KV4	QoQ	5.28	4.34	3.83	5.75	5.12	3.52	6.89	3.38	5.45	10.42	5.21
W4AxKV4	**FMPQ**	5.32	**4.31**	**3.82**	**5.73**	5.19	3.56	6.91	3.41	**5.43**	10.44	**5.17**

Table 2. Zero-shot accuracy evaluation on five common sense tasks for LLaMA-3 family models.

			Zero-shot Accuracy ↑					
Size	#Configuration	Method	PIQA	ARC-e	ARC-c	HellaSwag	Winogrande	Avg.
8B	FP16	Full Precision	79.9	80.1	50.4	60.2	72.8	68.6
	W8A8	SmoothQuant	79.5	79.7	49.0	60.0	73.2	68.3
	W4A16	Omniquant	78.4	77.9	48.5	58.8	72.7	67.2
	W4A8 KV4	QoQ	77.1	77.2	47.8	57.6	72.0	66.3
	W4AxKV4	**FMPQ**	77.5	76.7	47.5	**58.9**	72.1	66.5
70B	FP16	Full Precision	82.4	86.9	60.3	66.4	80.6	75.3
	W8A8	SmoothQuant	82.2	86.9	60.2	66.3	80.7	75.3
	W4A16	Omniquant	82.7	86.3	59.0	65.7	80.9	74.9
	W4A8 KV4	QoQ	81.4	85.7	58.4	64.9	79.9	74.0
	W4AxKV4	**FMPQ**	82.5	85.2	58.3	65.0	79.6	74.1

evaluations under different configurations, including FP16, W4A16 and W8A8, as the baseline systems. Moreover, we compare COMET with Qserve [29], a recent approach supporting W4A8KV4 LLM serving.

6.2 Algorithm Evaluation

Algorithm Benchmarks. We compare our proposed FMPQ algorithm with other baselines on the LLaMA-1 [50], LLaMA-2 [51], LLaMA-3 family models, as well as Mistral-7B [21], OPT-13B [62] and Qwen2-72B [2]. Following previous literature settings [4, 7, 12, 32, 48], we evaluated FMPQ-quantized models on language modeling and downstream zero-shot tasks. Specifically, we evaluated the perplexity of quantized models on WikiText2 [35], and evaluated the zero-shot accuracy on PIQA [5], ARC [8] (including ARC-e and ARC-c), HellaSwag [61] and WinoGrande [47] with lm_eval [13].

Algorithm Baselines. We compare FMPQ with widely used PTQ LLM quantization algorithms, including weight-only and weight-activation quantization methods. We use SmoothQuant [56] as the basic weight-activation quantization method, and also compare with weight-only quantization algorithms including GPTQ [12], AWQ [28] and Omniquant [48]. Additionally, we assess the QoQ algorithm, as implemented in Qserve, using a group-wise W4A8 KV4 configuration, with a group size of 128 and a single FP16 scale factor per group. Furthermore, we aggressively extend Omniquant to a full W4A4 quantization, to evaluate the corresponding accuracy degradation.

Perplexity Evaluation. Table 1 presents the evaluation results of Wikitext2 perplexity between FMPQ and other algorithm baselines. As one can notice, compared to W8A8 SmoothQuant and W4A16 Omniquant, FMPQ only introduces a slight perplexity increase (only 0.04 for LLaMA-1-30B and 0.07 for LLaMA-1-65B, respectively). When we further introduce KV cache quantization, the increased perplexity is as small as 0.05 on average, which is negligible. Moreover, unlike QoQ, which quantizes all activations to 8-bit, FMPQ selectively quantizes only about 16% of activations to 8-bit, allowing most GEMM tiles to be computed in W4A4 format. Specifically, for LLaMA-1-30B, only 8% of activations are quantized to 8-bit. These results indicate that FMPQ provides a practical solution for W4A4KV4 LLM serving. In comparison, when adopting a fully W4A4 Omniquant, the increased perplexity is unbearable (more than 5.21), hindering the practical deployment of quantized LLMs.

Zero-shot Accuracy. We further report the zero-shot accuracy of five common sense tasks in Table 2. Compared with the state-of-the-art W4A16 quantization method, the accuracy acquired by FMPQ is only decreased by 0.75%. For

LLaMA-3-8B, our FMPQ strategy even outperforms Omniquant when evaluating HellaSwag, demonstrating its efficiency. Additionally, our fine-grained mixed-precision quantization strategy consistently outperforms QoQ's W4A8 KV4 quantization on most downstream tasks.

6.3 Kernel Performance

We evaluate the COMET-W4Ax kernel across a range of GEMM workloads with batch sizes from small (2, 4, and 8) to large (16, 64, and 256), representing diverse use cases. We set the W4A4 ratio as 75% for the following kernel performance evaluations since it's the lower bound of the given kernel performance. Specifically, COMET often achieves a higher W4A4 kernel proportionality in practical model deployment. We compare against baselines including cuBLAS-W16A16 [1], TRT-LLM-W4A16, and TRT-LLM-W8A8 [40], with cuBLAS-W16A16 latency normalized to 1.

Small Batch Size Performance. Figure 9(a) presents the normalized performance of various GPU kernels with small batch sizes. The results show that COMET-W4Ax achieves average speedups of 1.48×, 1.25× and 1.37× over cuBLAS-W16A16, TRT-LLM-W4A16, and TRT-LLM-W8A8, respectively. Although TRT-LLM-W8A8 benefits from lower-precision computation, it only achieves a 1.09× speedup over cuBLAS-W16A16 due to a bottleneck in data loading rather than computation for small batch sizes. TRT-LLM-W4A16 outperforms W8A8 by effectively reducing weight data loading. In comparison, COMET-W4Ax demonstrates superior performance by addressing data loading challenges and enhancing computational efficiency.

Large Batch Size Performance. Figure 9(b) shows the normalized speedups of COMET-W4Ax compared to the baselines. As one can notice, COMET-W4Ax consistently delivers the best performance, especially in large-batch parallelism scenarios (e.g., batch size = 256). On average, COMET-W4Ax achieves speedups of 2.88×, 1.77×, and 1.33× over cuBLAS-W16A16, TRT-LLM-W4A16, and TRT-LLM-W8A8, respectively. As batch sizes increase to 64 and 256, TRT-LLM-W4A16 shows limited performance gains (only 1.10× and 1.38×) due to being compute-bound for large-batch cases. In contrast, TRT-LLM-W8A8 sees notable speedup improvements with larger batch sizes. In contrast to these kernels, which exhibit varying performance, COMET-W4Ax achieves substantial and consistent gains across different batch sizes, with speedup factors of 2.91×, 2.97×, and 2.75× for batch sizes of 16, 64, and 256, respectively.

Analysis on Varying Kernels. As Figure 9 shows, the performance gains for different GEMM shapes are quite different. For example, the performance remains relatively stable in some cases (e.g., 13.5K × 5K), while it varies significantly in others (e.g., 5K × 13.5K). This is because, in cuBLAS, the optimal tile partition varies for different GEMM shapes [1]. However, to accommodate mixed-precision tile partition, we fixed the tile partitioning strategy in the COMET-W4Ax kernel, which resulted in suboptimal performance gains in specific cases.

In summary, COMET-W4Ax achieves superior performance across both small and large batch sizes, highlighting the effectiveness and generality of our kernel design. Specifically, the performance gains for large batch sizes are promising and we recommend programmers to adopt the COMET kernel in large-scale LLM serving.

6.4 End-to-End Evaluation

We explore the maximum achievable throughput of different inference systems, within the same memory constraints on a single A100-80G-SXM4. Specifically, we adopt two different settings, including an input/output sequence length of 1024/512 and an input/output sequence length of 128/128, to evaluate models including Mistral-7B, LLaMA family models and Qwen2-72B. Furthermore, we also compare the normalized throughput under the same batch sizes for LLaMA-3-8B.

Throughput Evaluation. Figure 10 presents the relative throughput performance of different inference systems. We set the TRT-LLM-W4A16 as the baseline. According to our evaluation, COMET achieves 2.02× and 1.63× higher throughput on average for two different input/output sequence length settings, respectively. With the help of low-precision quantization on weight, activation and KV cache, we can easily support large-batch parallelism for large models such as LLaMA-3-70B and Qwen2-72B. Specifically, relative to the best-performing configurations (either W4A16 or W8A8), COMET demonstrates impressive performance improvement: it achieves $1.18\times - 1.93\times$ higher throughput for 7B and 8B models, 1.74× higher throughput for LLaMA-2-13B, 2.81× higher throughput for LLaMA-1-30B and $1.28\times - 3.27\times$ higher throughput for 70B and 72B models. COMET performs better when processing tasks with longer output sequences (512 tokens), as the proposed 4-bit KV cache can effectively migrate the bottleneck of large batch execution. Moreover, COMET achieves an average 1.17× speedup over Qserve. This improvement is primarily due to the efficient use of the INT4 tensor cores in the A100, as well as the reduction in dequantization cost.

Comparisons across Batch Sizes. We present the speedup results across different batch sizes for LLaMA-3-8B in Figure 11. As the batch size increases, the execution throughput gradually improves. For example, when setting the batch size as 64 for TRT-LLM-FP16, the throughput is increased by 7.52× compared with the batch size of 4. Hence, supporting large-batch parallelism is essential for improving the end-to-end throughput on modern GPUs. Following the setting in COMET, we can achieve even larger batch sizes. Additionally, under the same batch sizes, COMET consistently outperforms the best configurations of TensorRT-LLM. According to our evaluation, COMET achieves a 1.37× speedup than SOTA TensorRT-LLM configurations. This is primarily

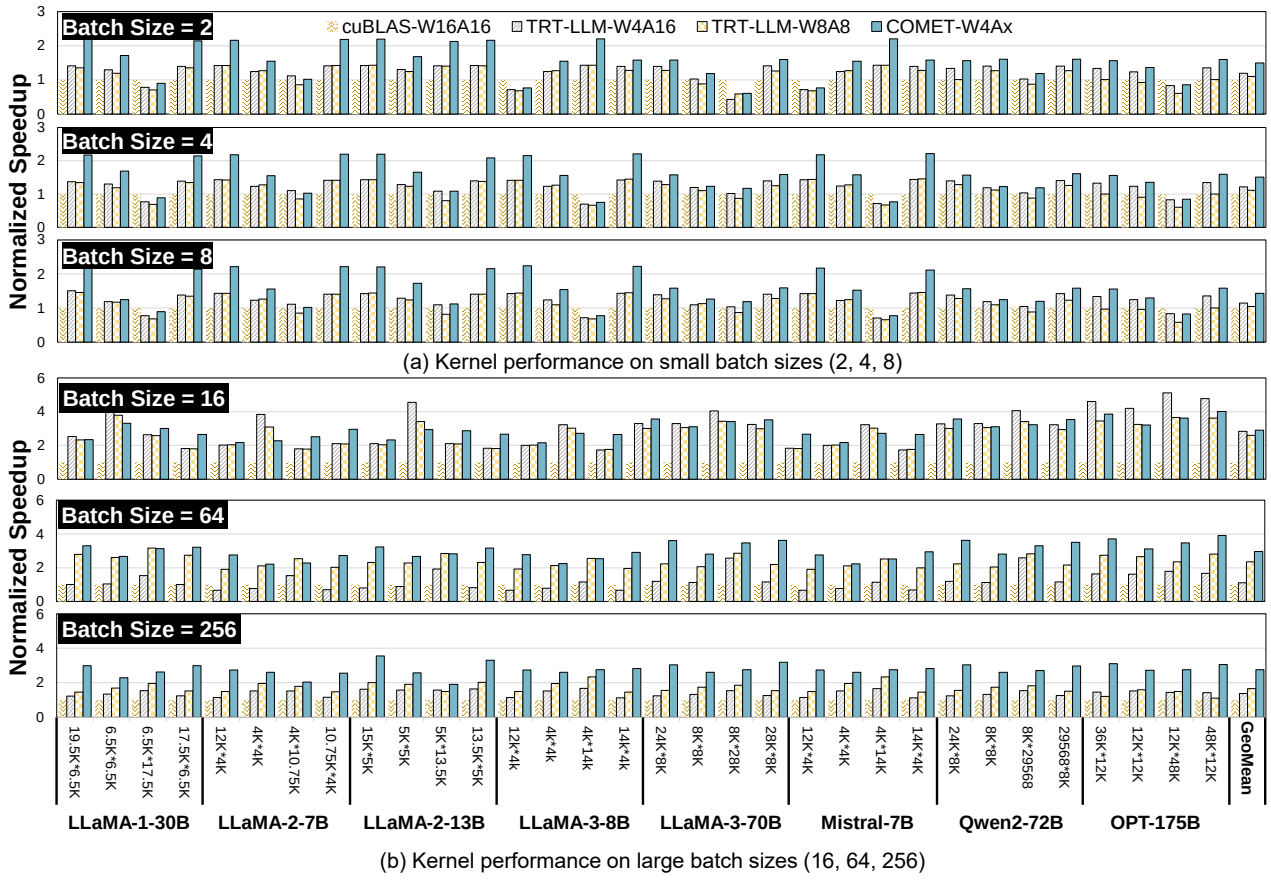


Figure 9. Kernel performance evaluation across different workloads and batch sizes.

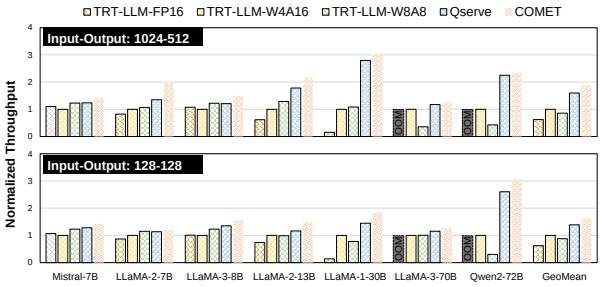


Figure 10. Compared to the SOTA inference systems TRT-LLM and Qserve, COMET provides higher throughput across various LLMs, ranging from 7B to 72B.

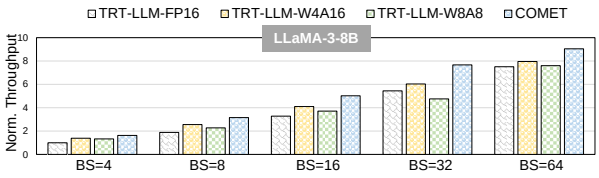


Figure 11. Throughput comparison between COMET and baseline inference systems for LLaMA-3-8B across batch sizes. We use an input-output sequence length of 1024-512.

due to our efficient utilization of INT4 tensor cores and fast dequantization.

Comparisons across Various LLMs. To demonstrate the generality of COMET on performance improvement, we further evaluate the end-to-end throughput across different LLMs at a batch size of 4, a typical configuration for small batches. As shown in Figure 12, COMET achieves an average 2.20× and 1.43× higher throughput than TRT-LLM-FP16 and TRT-LLM-W8A8, respectively. Due to the memory-bound nature of LLM inference with small batch sizes, TRT-LLM-W4A16 demonstrates a 1.16× performance improvement over TRT-LLM-W8A8. Nevertheless, when compared with the SOTA TRT-LLM-W4A16 configuration, COMET still offers a 1.18× throughput improvement without relying on increased batch parallelism. This improvement is due to COMET's effective use of high-efficiency low-precision

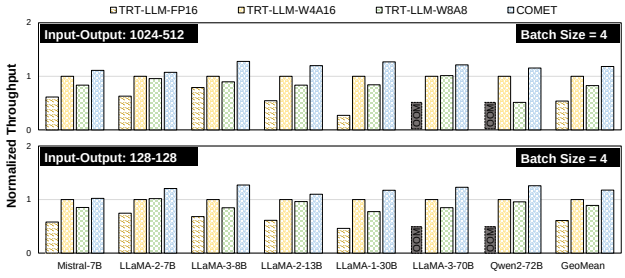


Figure 12. Normalized end-to-end throughput performance for various LLMs at a batch size of 4.

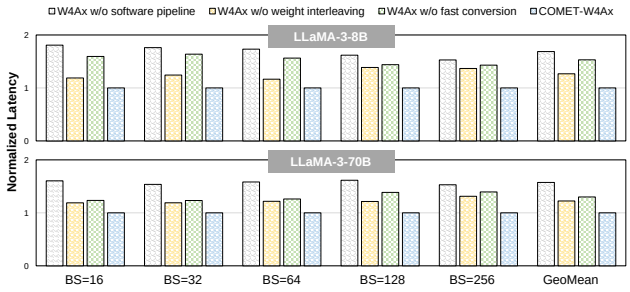


Figure 13. Ablation study on the proposed optimization strategies, including pipelining, weight interleaving and data conversion. The normalized kernel latency (lower is better) shows the effectiveness of our optimization techniques.

tensor cores (INT4 and INT8) and reduced dequantization overhead in mixed-precision quantization.

6.5 Ablation Study

Kernel Optimization. To support mixed-precision computation, we have developed a novel W4Ax kernel and employed various strategies to optimize the proposed COMET-W4Ax kernel, as detailed in Section 4. To evaluate the effectiveness of these optimizations, we conducted an ablation study on the LLaMA-3 models across batch sizes ranging from 16 to 256. We assessed different kernel versions, including W4A8, a naïve W4Ax kernel without software pipeline (W4Ax w/o software pipeline), a W4Ax kernel without weight interleaving (W4Ax w/o weight interleaving), a kernel without fast INT4to8 conversion (W4Ax w/o fast conversion), a kernel without mapping optimization (W4Ax w/o optimization), a kernel implementing the tile remapping (W4Ax w/ remapping) strategy proposed in Section 4.4, and a fully optimized COMET-W4Ax kernel. Additionally, we included the best-performing W4A4 kernel implemented with CUTLASS as an Oracle kernel, to analyze the theoretical upper bound of kernel performance. Note that full W4A4 quantization leads to significant accuracy degradation [48], making it impractical for LLM serving systems.

Figure 13 profiles the impact of SIMT-enhanced pipeline, weight interleaving, and data conversion optimizations on

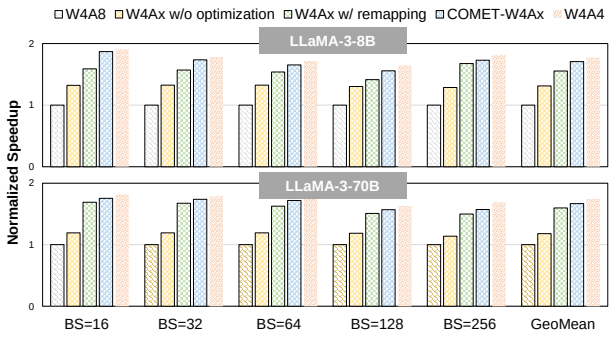


Figure 14. Performance improvement gained by different optimization strategies. A naïve implementation of W4Ax kernel can only achieve a limited performance improvement, while a highly optimized kernel can achieve a 1.69× speedup. COMET-W4Ax can achieve approximately 96% performance of the Oracle W4A4 kernel.

the designed W4Ax kernel performance. The results show that without the SIMT-enhanced software pipeline, weight interleaving, and fast conversion, COMET-W4Ax experiences performance degradations of 1.69×, 1.27×, and 1.53×, respectively. These optimizations collectively reduce the dequantization cost, making COMET-W4Ax significantly more efficient for mixed-precision GEMM computations.

Figure 14 illustrates the normalized performance differences between the COMET-W4Ax and the under-optimized W4Ax kernel. Compared to the W4A8 GEMM kernel, a naïve implementation of the W4Ax kernel achieves only 1.31× and 1.18× speedups. These potential gains stem from using INT4 tensor cores, which offer 2× higher throughput. However, due to the lack of load balancing across different SMs, the utilization of INT4 tensor cores falls short of expectations. By implementing tile and SM remapping, the speedup increases to 1.56× and 1.60×, respectively. By further eliminating the one-to-one binding between tiles and SMs, COMET-W4Ax achieves 1.71× and 1.67× speedup for GEMM computations of the LLaMA-3-8B and LLaMA-3-70B models, respectively. Additionally, COMET-W4Ax achieves 92.7% – 97.8% of the Oracle W4A4 kernel's performance, highlighting its broad applicability. The results also indicate that our fine-grained SM scheduling strategy effectively alleviates the imbalance of computing kernels between W4A4 and W4A8. It is worth noting that even an Oracle W4A4 kernel cannot achieve a 2× speedup over the W4A8 kernel, as GEMM computation performance on modern GPUs is constrained by factors such as GPU utilization and data flow, making theoretical peak performance unattainable in practice.

End-to-end Performance. We further analyze the impact of weight-activation and KV cache quantization on end-to-end throughput. In this analysis, COMET-W4Ax represents the application of weight-activation quantization only, while COMET-KV4 denotes KV cache quantization

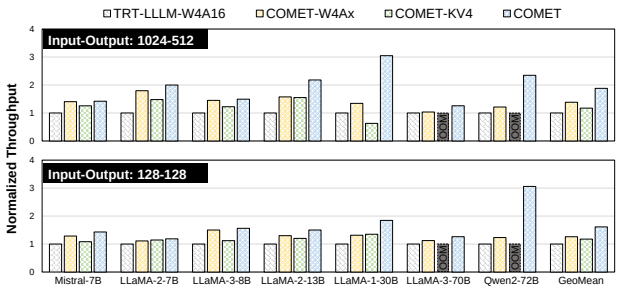


Figure 15. Ablation study on end-to-end performance, evaluating the individual effects of weight-activation quantization and KV cache quantization.

only within the COMET system. As shown in Figure 15, COMET-W4Ax achieves an average 1.32× performance improvements over TRT-LLM-W4A16 on diverse LLMs. Compared to weight-only W4A16 quantization, COMET-W4Ax leverages the high-performance INT4 and INT8 tensor cores to significantly enhance inference performance. When only adopting 4-bit KV cache quantization, COMET-KV4 yields a more limited 1.17× improvement over baseline, as it reduces KV cache storage costs but does not reduce computing costs. Additionally, KV cache only quantization cannot reduce the storage cost for weight parameters, limiting its effectiveness in deploying large-scale models such as LLaMA-3-70B and Qwen2-72B. By combining weight-activation and KV cache quantization, COMET maximizes modern GPU processing capabilities by both leveraging high-performance low-precision tensor cores and reducing storage costs. These optimizations work together to deliver a substantial boost in end-to-end inference performance, allowing COMET to achieve an average 1.82× increase in throughput.

7 Discussion & Future Wrok

COMET is a pioneering mixed-precision LLM inference framework that utilizes data distribution characteristics to achieve optimized end-to-end performance. Its current implementation focuses on optimizing the GEMM kernel, yielding significant speedups in throughput. Our evaluation shows that the proposed COMET-W4Ax kernel performs near the theoretical upper bound, demonstrating the effectiveness of COMET's design for mixed-precision computations.

Moving forward, we aim to further enhance COMET by incorporating attention kernel optimizations. In LLM inference, GEMM and attention computations occupy approximately 65% and 32% of the total runtime, respectively [33]. While COMET has substantially optimized GEMM performance, attention computation remains a critical component of the runtime. Previous studies have introduced efficient methods for optimizing attention during both prefill and decode phases [9, 17, 52], using algorithmic transformations to reduce data transfer costs. These attention optimizations operate independently of COMET-W4Ax and offer a promising next step for further performance gains.

In addition to attention kernel improvements, we plan to integrate COMET with various compilation and scheduling optimizations developed in LLM serving systems. Techniques such as operator-level pipelining and intra-device parallelism [3, 16, 23, 43, 57, 66, 67] complement COMET and could enhance end-to-end performance. Integrating these scheduling strategies with COMET's framework may unlock further efficiencies, making it a robust solution for high-performance LLM inference.

8 Conclusion

In this paper, we present COMET, the first mixed-precision LLM inference framework designed for practical W4A4KV4 LLM serving, primarily built on the proposed FMPQ algorithm and the COMET-W4Ax kernel. Specifically, the FMPQ algorithm efficiently achieves low-precision quantization for activations and the KV cache with minimal accuracy loss. Moreover, the open-source COMET-W4Ax kernel can be seamlessly integrated into existing inference systems. It includes optimizations for data layout, GPU software pipeline, and streaming multiprocessor scheduling, addressing data access and load imbalance issues in mixed-precision GEMM on modern GPUs. Evaluations on a single A100-80G-SXM4 demonstrate that COMET achieves a 2.02× end-to-end throughput improvement over state-of-the-art baselines, showcasing its potential for enhancing LLM inference efficiency.

Acknowledgments

We sincerely thank the anonymous reviewers for their insightful suggestions for improving this paper. This work was partially supported by the National Key R&D Program of China (Grant No. 2023YFB4404400) and the National Natural Science Foundation of China (Grant No. 62222411, 62025404). Ying Wang (wangying2009@ict.ac.cn) is the corresponding author.

References

[1] cublas docs, 2024. https://docs.nvidia.com/cuda/cublas/index.html.
[2] Qwen2 technical report. 2024.
[3] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.
[4] Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms. *arXiv preprint arXiv:2404.00456*, 2024.
[5] Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439, 2020.

[6] 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.

[7] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher M De Sa. Quip: 2-bit quantization of large language models with guarantees. *Advances in Neural Information Processing Systems*, 36, 2024.

[8] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

[9] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[10] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

[11] 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.

[12] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[13] Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, et al. A framework for few-shot language model evaluation. *Version v0. 0.1. Sept*, page 8, 2021.

[14] Georgi Gerganov. ggerganov/llama.cpp: Port of facebook's llama model in c/c++., 2023. https://github.com/ggerganov/llama.cpp.

[15] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[16] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

[17] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Yuhan Dong, Yu Wang, et al. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics. *Proceedings of Machine Learning and Systems*, 6:148–161, 2024.

[18] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

[19] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus. *Proceedings of Machine Learning and Systems*, 5:680–694, 2023.

[20] Shashank Mohan Jain. Hugging face. In *Introduction to transformers for NLP: With the hugging face library and models to solve problems*, pages 51–67. Springer, 2022.

[21] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[22] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. Mlir-based code generation for gpu tensor cores. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 117–128, 2022.

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[24] Changhun Lee, Jungyu Jin, Taesu Kim, Hyungjun Kim, and Eunhyeok Park. Owq: Lessons learned from activation outliers for weight quantization in large language models. *arXiv preprint arXiv:2306.02272*, 2023.

[25] Baolin Li, Viiay Gadepally, Siddharth Samsi, and Devesh Tiwari. Characterizing multi-instance gpu for machine learning workloads. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 724–731. IEEE, 2022.

[26] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 173–189, 2022.

[27] Yun Li, Lin Niu, Xipeng Zhang, Kai Liu, Jianchen Zhu, and Zhanhui Kang. E-sparse: Boosting the large language model inference through entropy-based n: M sparsity. *arXiv preprint arXiv:2310.15929*, 2023.

[28] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

[29] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. *arXiv preprint arXiv:2405.04532*, 2024.

[30] Shih-yang Liu, Zechun Liu, Xijie Huang, Pingcheng Dong, and Kwang-Ting Cheng. Llm-fp4: 4-bit floating-point quantized transformers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 592–605, 2023.

[31] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10012–10022, 2021.

[32] Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. Spinquant–llm quantization with learned rotations. *arXiv preprint arXiv:2405.16406*, 2024.

[33] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.

[34] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. In *Forty-first International Conference on Machine Learning*, 2024.

[35] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2016.

[36] NVIDIA. Cutlass 3.2, 2024. https://github.com/NVIDIA/cutlass.

[37] NVIDIA. Nsight compute profilling guide, 2024. https://docs.nvidia.com/nsight-compute/ProfilingGuide/#introduction.

[38] NVIDIA. Nsight system., 2024. https://developer.nvidia.com/nsight-systems.

[39] NVIDIA. Parallel thread execution isa version 8.5, 2024. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[40] NVIDIA. Tensorrt-llm, 2024. https://github.com/NVIDIA/TensorRT-LLM.

[41] OpenAI. Gpt-4 technical report, 2023.

[42] Muhammad Osama, Duane Merrill, Cris Cecka, Michael Garland, and John D Owens. Stream-k: Work-centric parallel decomposition for

dense matrix-matrix multiplication on the gpu. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 429–431, 2023.

[43] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[44] Jeff Pool and Chong Yu. Channel permutations for n: m sparsity. *Advances in neural information processing systems*, 34:13316–13327, 2021.

[45] 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.

[46] 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.

[47] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

[48] Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. Omniquant: Omnidirectionally calibrated quantization for large language models. In *The Twelfth International Conference on Learning Representations*, 2023.

[49] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. In *The Twelfth International Conference on Learning Representations*, 2023.

[50] 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.

[51] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[52] Tri, Dao and Daniel, Haziza and Francisco, Massa and Grigory, Sizov. Flash-decoding for long-context inference, 2023. https://pytorch.org/blog/flash-decoding.

[53] Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. Quip#: Even better llm quantization with hadamard incoherence and lattice codebooks. *arXiv preprint arXiv:2402.04396*, 2024.

[54] 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.

[55] BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

[56] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[57] Jiale Xu, Rui Zhang, Cong Guo, Weiming Hu, Zihan Liu, Feiyang Wu, Yu Feng, Shixuan Sun, Changxu Shao, Yuhong Guo, et al. vtensor: Flexible virtual tensor management for efficient llm serving. *arXiv preprint arXiv:2407.15309*, 2024.

[58] 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.

[59] Zhihang Yuan, Lin Niu, Jiawei Liu, Wenyu Liu, Xinggang Wang, Yuzhang Shang, Guangyu Sun, Qiang Wu, Jiaxiang Wu, and Bingzhe Wu. Rptq: Reorder-based post-training quantization for large language models. *arXiv preprint arXiv:2304.01089*, 2023.

[60] Yuxuan Yue, Zhihang Yuan, Haojie Duanmu, Sifan Zhou, Jianlong Wu, and Liqiang Nie. Wkvquant: Quantizing weight and key/value cache for large language models gains more. *arXiv preprint arXiv:2402.12065*, 2024.

[61] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4791–4800, 2019.

[62] 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.

[63] Yingtao Zhang, Haoli Bai, Haokun Lin, Jialin Zhao, Lu Hou, and Carlo Vittorio Cannistraci. Plug-and-play: An efficient post-training pruning method for large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

[64] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[65] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving. *Proceedings of Machine Learning and Systems*, 6:196–209, 2024.

[66] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.

[67] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, et al. Nanoflow: Towards optimal large language model serving throughput. *arXiv preprint arXiv:2408.12757*, 2024.