Quant-LLM: Accelerating the Serving of Large Language Models via FP6-Centric Algorithm-System Co-Design on Modern GPUs

Haojun Xia, *University of Sydney;* Zhen Zheng and Xiaoxia Wu, *Microsoft;* Shiyang Chen, *Rutgers University;* Zhewei Yao, Stephen Youn, Arash Bakhtiari, and Michael Wyatt, *Microsoft;* Donglin Zhuang and Zhongzhu Zhou, *University of Sydney;* Olatunji Ruwase, Yuxiong He, and Shuaiwen Leon Song, *Microsoft*

https://www.usenix.org/conference/atc24/presentation/xia

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Quant-LLM: Accelerating the Serving of Large Language Models via FP6-Centric Algorithm-System Co-Design on Modern GPUs

Haojun Xia, *University of Sydney*
Zhen Zheng, *Microsoft*
Xiaoxia Wu, *Microsoft*
Shiyang Chen, *Rutgers University*
Zhewei Yao, *Microsoft*
Stephen Youn, *Microsoft*
Arash Bakhtiari, *Microsoft*
Michael Wyatt, *Microsoft*
Donglin Zhuang, *University of Sydney*
Zhongzhu Zhou, *University of Sydney*
Olatunji Ruwase, *Microsoft*
Yuxiong He, *Microsoft*
Shuaiwen Leon Song, *Microsoft*

Abstract

Six-bit quantization (FP6) can effectively reduce the size of large language models (LLMs) and preserve the model quality consistently across varied applications. However, existing systems do not provide Tensor Core support for FP6 quantization and struggle to achieve practical performance improvements during LLM inference. It is challenging to support FP6 quantization on GPUs due to (1) unfriendly memory access of model weights with non-power-of-two bit-width and (2) high runtime overhead of weight de-quantization. To address these problems, we propose TC-FPx, the first full-stack GPU kernel design scheme with unified Tensor Core support of 6-bit and arbitrary bit-width quantization (5-bit, etc.). We integrate TC-FPx kernel into an existing inference system, providing new end-to-end support (called Quant-LLM) for quantized LLM inference, where better trade-offs between inference cost and model quality are achieved with 6-bit quantization. Experiments show that Quant-LLM enables the inference of LLaMA-70b using only a single GPU, achieving 1.69×-2.65× higher normalized inference throughput than the FP16 baseline. The source code is publicly available at https://github.com/usyd-fsalab/fp6_llm.

1 Introduction

Large Language Models (LLMs) [1, 29, 33–35, 41] are renowned for their capacity to process diverse language-related tasks [2, 8, 9, 28]. However, it is challenging to deploy LLMs as these models are also characterized by their expansive size, e.g., 175 billion parameter GPT-3 [1] and 1.76 trillion parameter GPT-4 [29]. On one hand, it requires large amounts of GPU memory (326 GB for GPT-3 in FP16) only to accommodate model weights, whereas an A100/H100 GPU [20, 21] only has up to 80 GB memory. On the other hand, LLM inference faces severe "memory wall" issues [12, 37] during token generation, where the speed of LLM inference is mainly limited by the time reading model weights from GPU DRAM. It makes LLM inference *memory bounded*, under-utilizing the computational power of GPUs.

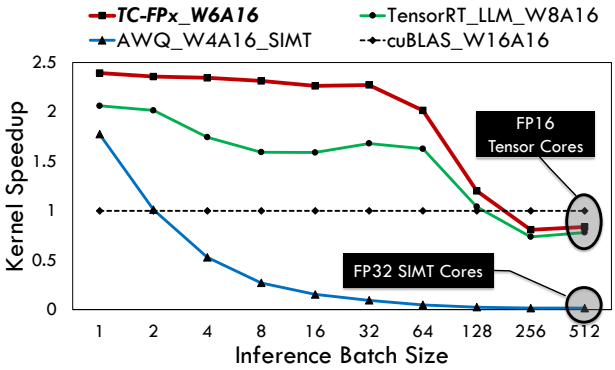


Figure 1: Performance of a linear layer within the llama-65b [33] model. The shapes of the weight/activation matrices are (8192, 22016) and (22016, Batch Size).

Model quantization [4, 6, 14, 32, 39, 42, 44] reduces both GPU memory footprint and DRAM data access. It uses fewer bits to represent each model weight, resulting in a more compact representation of the model. However, only a small set of bit-widths (i.e., 4-bit and 8-bit) are efficiently supported in existing systems [3, 14, 15, 26] on modern GPUs. We found that 6-bit is an overlooked "sweet spot" for LLM quantization, where superior trade-offs between inference cost and model quality can be achieved with FP6 quantization. We tried different bit-widths (goes from 16-bit baseline to 4-bit) to quantize the models, and the narrowest bit-width we can achieve is 6-bit, where the accuracy drop compared to FP16 is negligible constantly across various LLM models. This finding is consistent with recent studies [32, 36]. However, there is still no efficient system support for the 6-bit linear layer execution (i.e., matrix multiplication) on modern GPUs. Thus, it is urgent to develop the system support for 6-bit quantization fully leveraging the computing power of GPUs.

On one hand, more efficient LLM inference can be achieved with 6-bit quantization compared to larger-bit quantization (e.g., 8-bit). Firstly, more GPU memory can be saved, e.g. around 40 GB memory can be saved if deploying the GPT-3

model with 6-bit rather than 8-bit quantization. Secondly, LLM inference can be further accelerated as the time of reading model weights from GPU DRAM can be effectively reduced. As shown in Figure 1, the linear layer implemented with our newly proposed 6-bit quantization system design (TC-FPx_W6A16) is constantly faster (up to 1.45×) than the state-of-the-art support for 8-bit quantization (TensorRT_LLM_W8A16). On the other hand, 6-bit quantization can more effectively preserve model quality than smaller-bit quantization (e.g., 4-bit). Recent studies [10, 30] observe that the quality degradation of LLMs associated with 4-bit quantization [5, 14, 39, 40] can be significant. Besides, recent research [36] also demonstrates that in tasks extending beyond zero-shot[1] measurements, such as code generation and summarization, 4-bit methods underperform and lack robustness, whereas 6-bit quantization displays strong and consistent performance across these varied applications.

Motivated by the above observations, we propose TC-FPx, the first full-stack GPU system design scheme with unified Tensor Core [20,21] support of float-point weights for various quantization bit-width (6-bit, 5-bit, 3-bit, etc.), mitigating the "memory wall" issues during LLM inference. TC-FPx breaks the limitations of the underlying GPU hardware, allowing the GPU to support linear layer calculations involving model weights of arbitrary bit width. In TC-FPx, Tensor Cores are utilized for intensive computation of matrix multiplications, while SIMT cores are effectively leveraged for weight de-quantization, transforming the x-bit model weights to FP16 type during runtime before feeding them to Tensor Cores. To optimize GPU memory access, we propose *Ahead-of-time Bit-level Pre-packing* (Section 5.2) to resolve the challenge of unfriendly memory access for weights with irregular bit-width (Section 4.2.1), leveraging the static pattern of model weights. Besides, we propose *SIMT-Efficient GPU Runtime* (Section 5.3) to minimize the runtime overhead of weight de-quantization (Section 4.2.2). Last but not least, we present the software pipeline of TC-FPx kernel, where SIMT cores, Tensor Cores, and the GPU memory hierarchy cooperate efficiently with high performance.

We integrate TC-FPx kernel into a state-of-the-art inference system [19], providing new end-to-end support (called Quant-LLM) for quantized LLM inference, where better trade-offs between inference cost and model quality are achieved. Currently, Quant-LLM mainly supports 6-bit quantization (FP6) for popular LLMs such as LLaMA [33], OPT [41] with various sizes. Evaluations show that Quant-LLM enables the inference of LLaMA-70b using only a single GPU, achieving 1.69×-2.65× higher normalized inference throughput than the FP16 baseline. Besides, Quant-LLM improves the inference throughput of OPT-30b by 1.78×-4.51×.

In summary, we make the following contributions:

- We identify the significance and key challenges in supporting FP6 quantization on modern GPUs.
- We propose TC-FPx, the first full-stack GPU kernel design scheme with unified Tensor Core support of float-point weights with arbitrary bit-width, e.g. FP6, FP5.
- We provide new end-to-end inference support for quantized LLMs through the integration of TC-FPx, achieving better trade-offs between inference cost and model quality.
- We evaluate Quant-LLM on various LLM models and demonstrate that it substantially outperforms the baseline.

2 Background

2.1 Quantization of Large Language Models

Although large language models (LLMs) are known for their impressive performance, their large size also creates challenges for model deployment. Thus, model quantization [4,6,14,32,39,42,44] is commonly used for LLM deployment, resulting in a more compact representation of the model. *Weight-only quantization* [6, 14] only reduces the precision of model weights (e.g., INT8, using an 8-bit integer to represent each weight) while still using an FP16 value to represent each activation. The major targets to be quantized are the weights of **linear layers** (i.e., matrix multiplication), which account for more than 99% of the overall LLM weights. The activations can also be quantized during inference [4, 39]. In this paper, we describe the precision of **W**eights and **A**ctivations with the term "**W**x**A**y", where x/y denotes the bit-width of weights/activations. Besides, the process of "dequantization" refers to transforming the quantized weights back to FP16.

2.2 IEEE Standard for Floating-Point

The IEEE 754 float-point standard defines a binary format for representing real numbers. Each floating point number consists of three parts: the sign bit (S), the exponent bits (E), and the mantissa bits (M). The corresponding value f of a float-point number can be calculated via:

$$f = (-1)^S \times (1.M) \times 2^{E-bias}; \quad bias = 2^{len(E)-1} - 1 \quad (1)$$

Please refer to [11] for details, where special cases for values like infinity, zero, and NaN (Not a Number) are also defined.

2.3 Tensor Cores vs. SIMT Cores

SIMT cores [2] are responsible for **general-purpose** processing tasks in GPUs, which handle a wide range of instructions including integer operations, floating-point operations,

[1] Zero-shot means that the model is directly instructed to perform a task without any additional examples to steer it.

[2] Or referred to as CUDA cores on NVIDIA GPUs.

load/store operations, etc. SIMT cores execute scalar (or vector) instructions operating on individual (or vector) data elements. Tensor cores [20, 21] are **specialized hardware** designed for accelerating matrix multiplication. Tensor cores have 16.0×/14.8× higher FLOPS than SIMT cores on A100 [20]/H100 [21] GPUs. Besides, Tensor cores work at a coarse-grained granularity, e.g. performing a matrix multiplication between two FP16 matrices of shape 16 × 16 and 16 × 8 with a single *mma* (matrix multiply and accumulate) instruction.

3 Motivations

8-bit [4, 39] and 4-bit quantization [6, 14, 42] are widely applied schemes for the current post-training LLMs. However, we found that 6-bit is an overlooked "sweet spot" for LLM quantization, where superior trade-offs between inference cost and model quality can be achieved with FP6 quantization. We tried different bit-widths (goes from 16-bit baseline to 4-bit) to quantize the models, and we found that the narrowest bit-width we can achieve is 6-bit, where the accuracy drop compared to FP16 is negligible constantly across various LLM models. This finding is consistent with recent studies [32, 36]. Besides, NVIDIA recently announced FP6 Tensor Cores that would be added to their next generation of GPUs (NVIDIA Blackwell [27]) in the future, also indicating that FP6 matters.

(I) Lower inference cost than 8-bit quantization. Compared to the 8-bit quantization, the cost of deploying LLMs can be further reduced through more aggressive 6-bit quantization without a visible accuracy drop. On one hand, the size of LLM weights can be significantly reduced, nearly 2.7× smaller than the FP16 baseline. Less GPU memory is required to store model weights, thereby requiring fewer GPUs and reducing the serving cost of deploying LLMs. On the other hand, 6-bit quantization can also more effectively accelerate the inference of LLMs. Given that the LLM inference is usually *memory-bounded*[3] during token generation, faster LLM inference can be achieved through reducing GPU DRAM access of the model weights. As shown in Figure 1, the execution of the linear layer within llama-65b model [33] is consistently faster (up to 1.42× faster) with our newly proposed 6-bit quantization system design (TC-FPx_W6A16) compared to the state-of-the-art 8-bit quantization support (TensorRT-LLM_W8A16 [26]). Given that linear layers are the most time-consuming part of the large language models, this speedup will directly translate to performance improvements for end-to-end inference scenarios (See Section 7.3).

(II) Better model quality than 4-bit quantization. Although 4-bit quantization more aggressively reduces memory footprint and DRAM access, it unavoidably causes degradation in model quality [10, 30]. In contrast, near-lossless model compression can be achieved with 6-bit quantization. As shown in Table 1 and Table 2, FP6 displays strong and consistent performance across various tasks including code generation and zero-shot perplexity performance. It also shows high robustness across various model sizes, e.g., 1B, 13B, and 65B LLaMA [33] models. We also find that INT4 quantization heavily relies on *Fine-Grained Quantization (FGQ)* methods to maintain high model quality, whereas our FP6 quantization already works well on coarse-grained quantization. Note that the data points in Table 1 and Table 2 are picked from [36]. For more details, please refer to this paper. In conclusion, FP6 quantization is a practical alternative to further democratize the deployment of LLMs without significantly sacrificing model quality on complex tasks and various model sizes.

[3] When the execution is memory-bounded, it means that the rate at which data is transferred to or from the GPU's memory is the bottleneck, rather than the computational capabilities of the GPU cores.

Table 1: Zero-shot evaluations, averaging over five datasets including PTB [17], Wikitext [18], and C4 [31]. Metric: perplexity, **lower is better**.

Fine-grained Quantization	FP16 N/A	*FP6* ✗	INT4 ✓	INT4 ✗
LLaMA-1B [33]	**24.13**	*24.83*	564.73	288.22
LLaMA-13B [33]	13.16	***13.09***	14.19	14.13
LLaMA-65B [33]	**6.41**	*6.42*	6.61	7.17

Table 2: Code Generation in HumanEval-X (JavaScript) [43]. Metric: pass@1↑, **higher is better**.

Fine-grained Quantization	FP16 N/A	*FP6* ✗	INT4 ✓	INT4 ✗
CodeGeeX2-6B [43]	31.50	***31.61***	28.35	25.15
StarCoder-15B [13]	**33.67**	*33.6*	32.32	32.18
CodeLLaMA-34B [16]	**45.05**	*44.51*	43.22	43.45

4 Design Choices and Challenges

4.1 Design Choices

Although there is an increasing demand for high-performance support of post-training FP6 quantization, currently there is no such efficient FP6-centric system design available that enables the aforementioned trade-offs against 4-bit and 8-bit quantization. Specifically, existing supports for linear layers are mainly designed for data types whose bit-width is **a power of 2** (e.g., 4-bit, 8-bit, and 16-bit). Given that it is not clear how to support FP6 efficiently on modern GPUs, we illustrate two important design choices in this section.

Necessity in enabling Tensor Cores. We find it essential to support Tensor Cores when performing inference of quantized LLMs. For example, we have evaluated the performance of AWQ's [14, 15] pure SIMT-core execution on various batch sizes to test its scalability. As shown in Figure 1, the runtime performance of linear layers without Tensor Core support (AWQ_W4A16_SIMT) becomes extremely low as the inference batch size increases. The reason behind this is twofold.

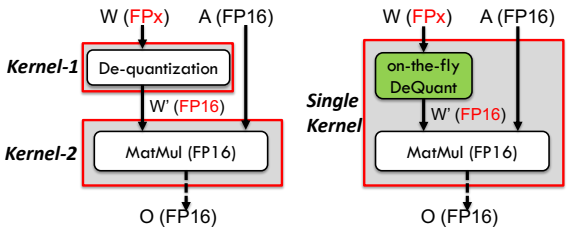


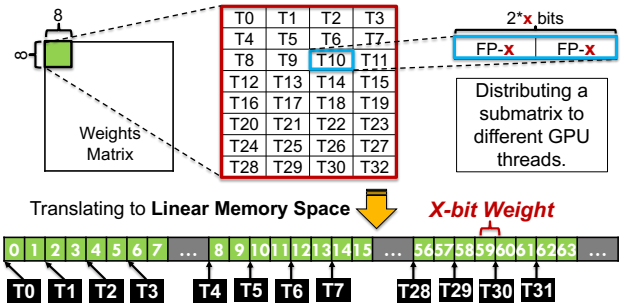
Figure 2: Two different methods to support weight-only WxA16 quantization during LLM inference. (Left) Dual kernels. (Right) Unified kernel.

On one hand, traditional SIMT cores are an order of magnitude slower than Tensor Cores for linear layer execution as described in Section 2.3. On the other hand, a large fraction of the SIMT core's computational power will be used to de-quantize the model weights at runtime, which further reduces the available computational power of SIMT cores for computing matrix multiplication. This motivates us to enable tensor cores for intensive computation of matrix multiplication while leveraging versatile SIMT cores for weight de-quantization.

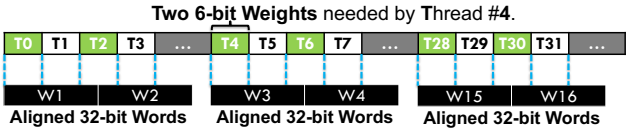
Unified kernel solution rather than dual kernels. The unique character of *WxA16 quantization* is that the activation matrices use FP16 but the weight matrices are stored in a narrower bit-width. However, Tensor Cores require both the weights and activations matrices to be stored in the same data type, e.g. FP16/INT8/INT4. The straightforward solution (i.e., dual kernel solution) adds an extra GPU kernel that de-quantizes the weights to FP16 before calling the normal FP16 kernel. However, such inference speed would be even slower than that of the model without quantization. As shown in Figure 2 (Left), two GPU kernels will be launched for the linear layer execution, and the de-quantized FP16 weights will be written to GPU DRAM before being read by the second GPU kernel, resulting in 2× DRAM access. It is more efficient to fuse the de-quantization and the matrix-multiply process into a single GPU kernel, eliminating the read/write of the de-quantized weights (W' in FP16).

4.2 Design Challenges

Given the design choices in Section 4.1, it is challenging to design a unified GPU kernel supporting FP6×FP16 matrix multiplication on modern GPUs. On one hand, modern GPU memory systems do not naturally support **irregular bit-width** (not a power of 2) because the minimal access size of GPU global/shared memory is 8/32 bits per thread and the memory addresses to access must be aligned. The complex data layout requirement of Tensor Cores makes it even more challenging for irregular bit-widths. On the other hand, the de-quantization computation is expensive as it requires a large amount of complex bit-level operations. Thus, how to fuse the de-quantization into the linear layer computation without hurting the overall performance is also non-trivial.



(a) Required Data Layout of Tensor Cores Input. T0 Means Thread #0.



(b) Accessing 6-bit weights at the granularity of 32-bit Words.

Figure 3: Memory Access of X-bit Weights for each Thread.

4.2.1 Hardware-Unfriendly Memory Access

During the execution of linear layers on modern GPUs, model weights should be loaded from DRAM to *registers* before the corresponding multiplication calculations can be performed. Usually, the model weights are loaded in two steps, to hide the high access latency of DRAM for high performance. Specifically, model weights are first loaded from GPU DRAM and buffered into on-chip memory (e.g., *shared memory*) for data reusing. After that, the buffered weights are then read from *shared memory* to *registers* for the actual computation.

Given that each GPU thread **cannot** directly access other threads' *registers*[4], each thread must put the model weights that are needed by itself to its private *registers* **on its own**. This process can become extremely challenging when the weights are stored with irregular bit-width (not 2^n, e.g., 6 bit), given the rigid data layout requirements of Tensor Cores. As shown in Figure 3a, the minimal input of FP16 Tensor Cores is a 8×8 sub-matrix in modern GPU architecture, and each GPU thread should hold a pair of weights in its *register*. In normal cases, each weight is stored with 16 bits, and each pair of weights can be naturally read from *shared memory* at the granularity of 32-bit words. However, each weight is stored with x-bits in our work[5], which makes memory access extremely unfriendly to modern GPU memory hierarchy.

On-chip Memory Access with Unused Bits: We use 6-bit quantization as an example to show the inefficiency in accessing weights with irregular bit-width. As shown in Figure

[4]Each GPU thread is allocated and owns a distinct portion of the whole registers available on GPU processors.

[5]Our design principles support not only 6-bit but also any other bit widths.

3b, weights are already buffered in *shared memory*, and each GPU thread needs to read a pair of weights (**12 bits**, $2*6bits$) from *shared memory*. However, *shared memory* has 32 memory banks and each memory bank outputs a **32-bit** word per memory request on GPU. Thus, a large fraction of bits read from shared memory will be unused, resulting in a significant waste of shared memory bandwidth. For instance, T0 (Thread #0) in Figure 3b only needs 12 bits. However, a 32-bit word (W1) will be read, resulting in 20 out of 32 bits (62.5%) unused. The waste of unused bits can get even more severe due to the requirement of **aligned memory access**[6] in modern GPU memory hierarchy. As shown in Figure 3b, the bits needed by T2 (Thread #2) are distributed in both W1 and W2. Thus, T2 needs to read both W1 and W2, reading $2*32$ bits from *shared memory*. However, only $6*2$ bits will be eventually used, resulting in 52 out of 64 bits (81.25%) unused and wasted. It is also worth noting that the memory management and access on GPU DRAM and *registers* suffer from similar problems due to the irregular bit-width.

4.2.2 High Computation Overhead of De-quantization

The runtime overhead of FPx-FP16 de-quantization can be extremely high, which easily slows down the overall execution. On one hand, large amounts of model weights need to be de-quantized at runtime, e.g. 70 billion FPx weights should be de-quantized for each LLM decoding step[7] for LLaMA-70b [34] inference. On the other hand, the runtime overhead to de-quantize each FPx weight is high, requiring complex bit-wise operations. According to Equation 2, new *exponent (E)* and *mantissa (M)* need to be calculated during runtime, to obtain the FP16 with the equivalent value of a given FPx.

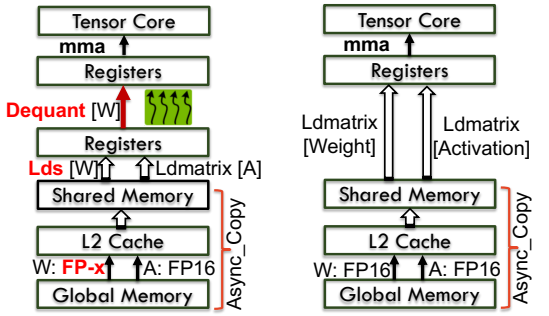
$$2^{E^{fp16}-bias^{fp16}} \times (1.M^{fp16}) = 2^{E^{fpx}-bias^{fpx}} \times (1.M^{fpx}) \quad (2)$$

In Equation 2, $bias^{fp16} = 15$ and $bias^{fpx} = 2^{len(E^{fpx})-1} - 1$. The sign field of the FP16 is identical to that of the FPx, and the mantissa of the FP16 can also be calculated by padding zeros to that of the FPx. What's more, the exponent of FP16 should be $E^{fp16} = E^{fpx} + bias^{fp16} - bias^{fpx}$, which is more computationally expensive. In summary, how to de-quantize FPx values efficiently also becomes a major challenge.

5 Design Methodology

We first provide an overview of our unified designs supporting various quantization bit-width with GPU Tensor Cores in Section 5.1. To solve the challenge of unfriendly memory access (Section 4.2.1), we propose *Ahead-of-time Bit-level Pre-packing* in Section 5.2. To deal with the challenge of the high computational overhead of de-quantization (Section 4.2.2),

[6]Memory access must be aligned, i.e., its address is a multiple of its size.
[7]To generate a sequence with n tokens, n-1 decoding steps are required.



(a) TC-FPx Design. (b) Traditional GEMM Design.

Figure 4: Design Overview.

we presented our designs to achieve *SIMT-Efficient GPU Runtime* in Section 5.3. At last, we presented our software pipeline designs in Section 5.4, where SIMT cores, Tensor Cores, and GPU memory hierarchy can work collaboratively.

5.1 Overview

Figure 4 compares TC-FPx, our x-bit weight-only quantized linear layer kernel design, with the traditional design for general-purpose matrix multiplication (GEMM) where both input matrices are in FP16. The model weight is stored with a reduced number of bits for TC-FPx. Consequently, an additional de-quantization stage (Dequant W) is introduced at the register level, where the FP6 weights are de-quantized to FP16 locally within each thread using SIMT cores. It is worth noting that the FP16 weights are not written back to *shared memory* but stored in *registers* for future use, eliminating unnecessary round-trip access to *shared memory*. Another difference is that TC-FPx loads x-bit weights from *shared memory* to *registers* using fine-grained `lds` (load shared) instructions instead of using the coarse-grained intrinsic `ldmatrix` (load matrix), which has a strict layout requirement and is less flexible.

5.2 Ahead-of-time Bit-level Pre-packing

As described in Section 4.2.1, memory access to weights with irregular bit-width is unfriendly to modern GPU memory hierarchy. To address this problem, we propose the insight that we can combine the memory read of *every 32 x-bit weights*, resulting in *x request of 4-byte word* per GPU thread. In this case, all the memory access would be aligned at the granularity of 32-bit words rather than the irregular bit-width. However, it is **not trivial** to combine the memory read of weights due to Tensor Cores' rigid data layout requirements, where the weights needed by each thread are not stored in continuous memory space.

To solve this problem, we propose to optimize the runtime memory access pattern during model serving by reorder-

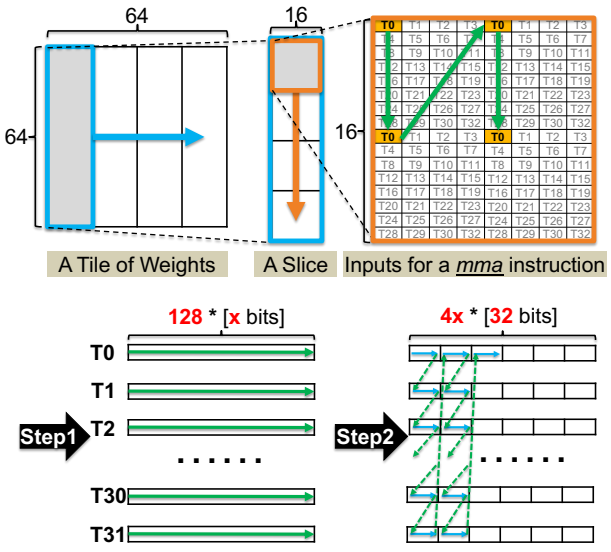


Figure 5: Ahead-of-time Bit-level Weight Pre-packing. This technique is independent of the actual bit-width (denoted as x) of model weights and can be applied to arbitrary bit-width.

ing the weights within each weight matrix and pre-pack the weights ahead of time, **leveraging the static pattern of model weights**. As model weights are statically determined after the model is trained and quantized, complicated memory layout transformation can be applied to the weights ahead of time, introducing no runtime overhead. Besides, this pre-packing process is a once-for-all overhead before model deployment, which usually takes several minutes to process a model.

In general, weight pre-packing consists of two steps. In the first step, we gather all the weights needed by each GPU thread and combine these weights locally. Given that the weights needed by each GPU thread are not originally in continuous locations (see Figure 3a) within each weight matrix, we must pick the weights for each GPU thread carefully. The weights picked for each thread are then combined locally in relative temporal order as they are consumed by Tensor Cores at runtime. In the second step, we combine all the weights needed by the whole GPU WARP (consisting of 32 GPU threads) into a unified linear memory space, in which order the weights will be stored in GPU DRAM before runtime. To fully eliminate *shared memory* bank conflict[8], we propose to combine the 32-bit word of each thread in a "jagged" order.

Step 1: Per-thread Weight Gathering Figure 5 demonstrates the weights picked by T0 (Thread #0) and the order to combine them. We suppose the WARP-level tiling size is 64×64, which means each weight matrix is divided into 64×64 data tiles and loaded to GPU's *shared memory* at this granularity for each WARP. Each weight tile is then further divided into four slices, as the weights are loaded from *shared memory* and used for Tensor Core computations on a slice-by-slice basis. What's more, each slice is divided into four 16×16 chunks, as Tensor Core processes 16×16 data items in each instruction. Within each 16×16 chunk, four pairs of FPx weights are picked for T0 and combined. As shown in Figure 5, we get 32 (i.e., the WARP size) groups of FPx weights after Step 1. The weights are combined and stored continuously within each group and each group of weights will be consumed by a certain GPU thread. In summary, each 64×64 weight tile is eventually assigned to 32 threads (a WARP), and each thread will consume 128 x-bit weights.

Step 2: Bit-level Assembling per WARP In Step 2, we assemble all the weights of different groups into a unified memory space. During this process, we consider the combined weights as continuous data to copy, temporarily ignoring the meaning of each bit. Specifically, 128 x-bit items are considered 4x items with 32 bits. Besides, we propose to assemble the weights of all groups in the **jagged order** shown in Figure 5. To begin with, the first 32-bit item of each thread is concatenated together. After that, the second 32-bit item of each thread is concatenated and appended to the previous results. By repeating this process, all weights can be stored continuously in a linear memory space and well-aligned (128-byte aligned). In this way, all weights can be simply copied from DRAM to *shared memory* at the granularity of 128-byte blocks without any changes, easily achieving optimal DRAM access. Besides, these weights can then be loaded from *shared memory* with optimal performance as well, where a WARP of threads read consecutive 32-bit items in *shared memory* for each memory request, fully avoiding bank conflict.

5.3 SIMT-Efficient GPU Runtime

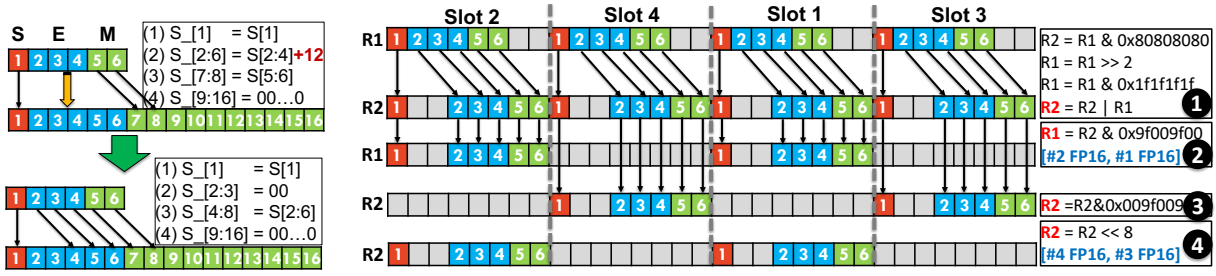
Parallel De-quantization To reduce the runtime overhead of FP-x weight de-quantization, we implemented FP-x de-quantization with optimized bit-wise SIMT core instructions. Besides, we propose to de-quantize multiple FPx weights in parallel, further reducing the SIMT overhead by 4× by exploiting the bit-level parallelism within each 32-bit register.

(1) Optimized Bit-wise Operations: As described in Section 4.2.2, the exponent for FP16 should be $E^{fp16} = E^{fpx} + bias^{fp16} - bias^{fpx}$, when casting an FPx to the equivalent FP16. To simplify this process, we adopted the mathematical transformation in [36], calculating the exponent of FP16 with $E^{fp16} = E^{fpx}$ instead. To maintain correctness, the result FP16 is then multiplied with the FP16 constant $2^{bias^{fp16} - bias^{fpx}}$:

$$cast(W_{fpx}) = new\_cast(W_{fpx}) \times 2^{bias^{fp16} - bias^{fpx}}. \quad (3)$$

Fig.6a shows the optimized FP16 to FP6 conversion. Although we only draw the cast from FP6 to FP16 for demon-

[8] Bank conflicts occur in shared memory when multiple threads access data in the same memory bank simultaneously, leading to lower throughput.



(a) Optimized FP6 to FP16 cast.

(b) 4-Way parallel de-quantization within 32-bit registers.

Figure 6: SIMT-Efficient Parallel De-quantization. To maintain correctness, the de-quantized FP16 should be multiplied with the FP16 constant $1.0 * 2^{bias^{fp16} - bias^{fpx}}$ after the process demonstrated in (b). Besides, we do not pad FPx values at global/shared memory level. The initial bit layout shown in (b) is only an intermediate representation in registers generated by the "Weight Split and Stitching" process during runtime. More importantly, this design scheme can also be applied to other bit widths accordingly, e.g., 5-bit. Four-way de-quantization might not be applicable for some bit widths, but two-way parallelism can always be used.

stration, it can be applied to any bit-width. The sign field of FP16 is identical to that of FPx. Besides, the lower bits of the exponent field and the higher bits of the mantissa field can be copied from FPx to FP16 together for efficiency. What's more, other bits of FP16 should be padded with zeros.

With careful designs, we succeeded in achieving cast from FP6 to FP16 with only two bit-wise *"and"*, one *"shifting"*, and one *"or"* as shown in ❶ of Figure 6b. The sign field is copied from FP6 to FP16 with the first *"and"* and all other bits of the FP16 are initialized to zeros at the same time eliminating the need to pad zeros to the exponent and mantissa fields later. All bits of the FP6 are then shifted right with the bit-wise *"right shifting"*. After that, the lower bits of the exponent and the higher bits of the mantissa in FP6 are first selected via the *"and"* between the FP6 and the bit mask "0x1f1f1f1f", and then copied to the FP16 with the bit-wise operation *"or"*.

(2) Bit-level Parallelism: Given the insight that we can exploit the bit-level parallelism within each 32-bit word, we propose to de-quantize multiple FPx weights in parallel, further reducing the runtime overhead of de-quantization. The detailed design is demonstrated in Figure 6b using FP6 as an example. The 32-bit registers are treated as four processing slots, where each slot works independently with the same instruction but different input FP6 data. Before the start of de-quantization, four FP6 should be stored in $R1$ (Register #1) with the initial data layout shown in the figure. With the code snippet ❶, these four FP6 can be simultaneously de-quantized into four FP16, where only the first 8 bits of each FP16 are stored in $R2$. After that, the first and the second FP16 are extracted to $R1$ with their last 8 bits padded with zeros, with the code snippet ❷. Finally, with the code snippet ❸ and ❹, the third and the fourth FP16 are extracted to $R2$.

Weight Split and Stitching We will then demonstrate the method to efficiently reconstruct the 6-bit weights from the 2+4 scheme [36] on GPUs with a carefully designed memory layout. Note that all the techniques discussed in the following paragraphs can also be applied to other bit-width, e.g., 5-bit weights can be decomposed into 1+4 scheme, and 7-bit weights can be decomposed into 1+2+4 scheme. With this method, the model weights with irregular bit-width could be transformed into several segments with regular bit-width (power of 2). Given that these segments have regular bit widths, the stitching process ("Runtime Weight Stitching") can be highly efficient with the following designs.

(1) Ahead-of-time Weight Split: To store the weights in a well-aligned manner in GPU's 32-bit *registers*, we split each weight into several segments, where the bit-width of each segment is 2^n, e.g. each 6-bit weight can be split into either 2+4 or 4+2. Based on this scheme, the index calculations for the following designs are significantly simplified. Note that the techniques described in Section 5.2 can be applied to any bit-width, thus the 2-bit and 4-bit segments can be pre-packed separately and efficiently according to Section 5.2.

(2) Runtime Weight Stitching: Before the de-quantization, the weights are first loaded from *shared memory* to *registers*. As each weight is split into several segments, the complete weights need to be reconstructed at the register level during runtime. To reduce this runtime overhead, we propose to extract and stitch the weights in parallel. As shown in Figure 7, two sets of registers are used to store 32 FP6 weights, where $Frag1\_PTR$ points to two 32-bit registers containing 32 2-bit segments while $Frag2\_PTR$ points to four 32-bit registers containing 32 4-bit segments. With our parallel stitching, four FP6 weights are reconstructed simultaneously, reducing the number of SIMT core instructions by 4×. As shown in Figure 7, four 2-bit segments are first extracted to Register #1 (❶), and four 4-bit segments are then extracted to Register #2 (❷). After that, Register #2 is right-shifted (❸) and its valid bits are copied to Register #1 (❹), resulting in complete 6-bit weights.

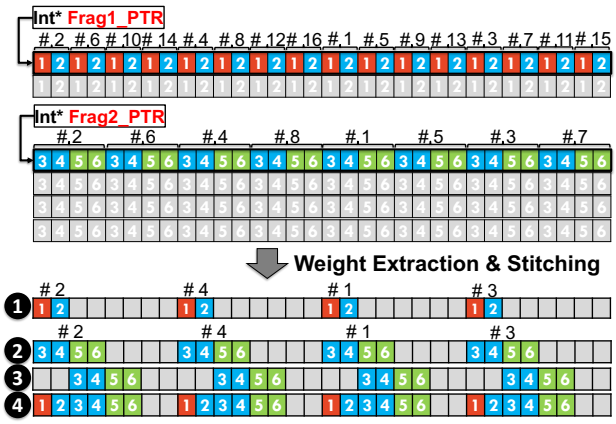


Figure 7: Parallel Weight Stitching. We show the stitching of 2-bit and 4-bit segments here, and the support of 1-bit segments can be designed accordingly. Eventually, weights with arbitrary bit widths can be effectively re-constructed by composing the segments with regular bit widths in runtime.

(3) Bit Reordering: To extract and stitch the weight in parallel, it is necessary to enforce the initial data layout in Figure 7. The key observation is that each four continuous segments must be placed in the order shown in the figure, e.g. the first four segments must be stored in the order of #2, #4, #1, and #3. Besides, the stride between each pair of 2/4-bit segments should be 6/4, respectively. Otherwise, it is not possible to stitch four segments simultaneously with only four SIMT core instructions. To satisfy the initial data layout requirements in Figure 7, we propose to ensure this layout via reordering the weight segments before runtime with no runtime overhead. Besides, this technique is supposed to be superimposed on the technique described in Section 5.2 as an additional pass.

Overall Pseudo Code Algorithm 1 shows the pseudo code for FP6[9], including both *Parallel De-quantization* and *Weight Stitching*. All the input and output variables in the pseudo code are stored in *registers*. As demonstrated in Figure 7, Algorithm 1 de-quantizes 32 FP6 weights in total, where four FP16 weights are generated for each loop. The transformations in Figure 7 (❶, ❷, ❸, and ❹) are achieved with the SIMT core operations of lines 5-8 in Algorithm 1. Eventually, the output *register* array (*OutputReg*) will be directly used by Tensor Cores as inputs.

5.4 Software Pipeline Design

In our designs, Tensor Cores are mainly used for matrix multiplications, while SIMT Cores are used for bit-level operations related to weight de-quantization and stitching (See Section 5.3). In this section, we will describe the appropriate timing of doing de-quantization so that Tensor Cores, SIMT Cores, and the GPU memory hierarchycan work in parallel without harmful instruction stalls due to data or barrier dependency.

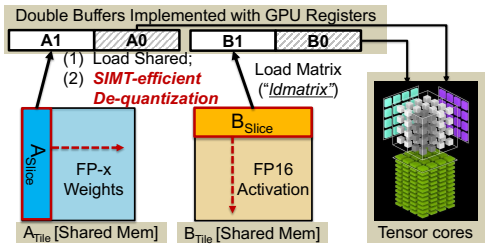
[9]Please refer to our complete source code on GitHub if you want to see our support of other bit-widths, e.g., FP5.

Algorithm 1 Weight Stitching & De-quantization.

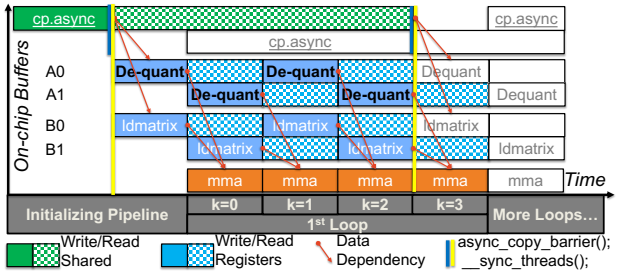
1: **Inputs:** int $Frag1\_ptr[]$, int $Frag2\_ptr[]$, half $Scales[]$
2: **Output:** int $OutputReg[]$
3: **for** int $i = 0$; $i < 8$; $i++$ **do**
4: //Weight Extraction & Stitching
5: unsigned int $R1 = (*Frag1\_ptr)\&0xc0c0c0c0$; ▷ ❶
6: unsigned int $R2 = (*Frag2\_ptr)\&0xf0f0f0f0$; ▷ ❷
7: $R2 = R2 >> 2$; ▷ ❸
8: $R1 = R1|R2$; ▷ ❹
9: //Advancing to next register or shifting current register.
10: **if** $i\%4 == 3$ **then**
11: $Frag1\_PTR++$;
12: **else**
13: $(*Frag1\_PTR) = (*Frag1\_PTR) << 2$;
14: **if** $i\%2 == 1$ **then**
15: $Frag2\_PTR++$;
16: **else**
17: $(*Frag2\_PTR) = (*Frag2\_PTR) << 4$;
18: //4-Way Parallel de-quantization.
19: $*R2 = *R1\&0x80808080$;
20: $*R1 = *R1 >> 2$;
21: $*R1 = *R1\&0x1f1f1f1f$;
22: $*R2 = *R2|*R1$;
23: $*R1 = *R2\&0x9f009f00$;
24: $*R2 = *R2\&0x009f009f$;
25: $*R2 = *R2 << 8$; ▷ R1 and R2 now each has 2 FP16 weights.
26: //Multiplying with the FP16 constant to maintain correctness.
27: $R1 = Multiply(R1, 1.0 * 2^{bias^{fp16} - bias^{fpx}})$;
28: $R2 = Multiply(R2, 1.0 * 2^{bias^{fp16} - bias^{fpx}})$;
29: //Multiplying with quantization scales & Output to registers.
30: $OutputReg[i*2] = Multiply(R1, Scales[i/2*2])$;
31: $OutputReg[i*2+1] = Multiply(R2, Scales[i/2*2+1])$;

Slice-by-slice De-quantization Instead of de-quantizing all the weights at once, we de-quantize the FPx weights slice by slice. As shown in Figure 8a, we assume that an FPx weights tile and an FP16 activation tile are already copied from DRAM to *shared memory*. The whole tile of weight in *shared memory* is then de-quantized in several steps. In each step, only a slice of FPx weights is loaded from *shared memory* to *registers*, de-quantized into FP16 weights with *SIMT-Efficient GPU Runtime* (Section 5.3), and then stored in the register buffer $A1$ or $A2$ as inputs for Tensor Cores. A_{Slice} and B_{Slice} are then multiplied using Tensor Cores.

Compared to de-quantizing the whole tile at once, our slice-by-slice de-quantization reduces the number of registers required to store the FP16 weights by 4×, significantly reducing register pressure. Besides, more opportunities are created for instruction-level parallelism, since Tensor Cores can be used immediately for computations once a slice of weights is de-quantized, rather than waiting for the entire tile.



(a) Slice-by-slice De-quantization.



(b) Space-time Diagram of the Kernel Pipeline.

Figure 8: Software Pipeline of TC-FPx GPU Kernel.

Effective Overlapping The software pipeline is illustrated via the space-time diagram in Figure 8b, where SIMT cores (working on de-quantization), Tensor Cores (working on matrix multiplication), and GPU memory hierarchy work collaboratively, achieving high instruction-level parallelism.

Firstly, global memory read is executed asynchronously using the *cp.async* [20] intrinsic, fully overlapped with other operations. Memory barrier and thread block synchronization are issued after the third slice is processed (at the end of k=2), making sure that the data for the next main loop is ready in *shared memory* so that the *"De-quant"* (de-quantization) and the *"ldmatrix"* operations can be started when k=3.

Secondly, shared memory read is also overlapped with tensor core operations. When the i_{th} slice is being computed, the data of the $(i+1)_{th}$ slice are read from *shared memory* simultaneously via *"De-quant"* and *"ldmatrix"*.

Last but not least, the SIMT core operations for weight de-quantization are also effectively overlapped with Tensor Core operations. Within the *"De-quant"* process of the i_{th} slice, the FPx weights are first loaded from *shared memory* to *registers* using the hardware intrinsic *load shared (LDS)*, and then immediately de-quantized into FP16 weights with SIMT cores. At the same time, Tensor Cores are computing the $(i-1)_{th}$ slice with no data dependency.

6 Implementation

We implemented the TC-FPx kernel supporting matrix multiply $C = A \times B$, where A is the weight matrix of shape $[M,K]$ and B is the activation matrix of shape $[K,N]$. The weight matrices are stored in our customized format described in Section 5.2, and the input and output activation matrices are stored in column-major. Thus, our TC-FPx kernel could be a drop-in replacement of cuBLAS kernels in inference frameworks for quantized LLMs. Our GPU kernel is implemented with more than 1.2K lines of CUDA codes on top of the code of Flash-LLM [38]. Our code is fully templated and can support different combinations of "eXmY"[10]. In this paper, we tested the performance of our TC-FPx kernel with model weights in FP6_e3m2 and FP5_e2m2. Meanwhile, the activations are always in FP16 format. Our TC-FPx kernels could be compiled separately into a *.so* dynamic link-able library, and we provide a set of C++ APIs to call the kernels. Our kernels could also be easily installed locally with "pip" (the package installer for Python) and called via our Pytorch APIs. Thus, our kernels could be easily used and integrated. Besides, we also provided C++/Pytorch APIs to pre-pack the weight matrices (See Section 5.2). More importantly, we provide new system support for end-to-end inference of quantized LLMs, by integrating our kernel into the state-of-the-art inference framework DeepSpeed [19].

7 Evaluation

We evaluate the performance at two levels: kernel-level benchmarking using TC-FPx GPU kernels and model-level end-to-end inference using DeepSpeed integration (which we call Quant-LLM). The kernel-level evaluation is conducted on the NVIDIA A100-40GB platform with CUDA 11.8, and we mainly evaluate the performance of linear layers within LLMs during the token generation phase. The utilization of each GPU hardware unit during runtime (Section 7.1) is measured using NVIDIA Nsight Compute [23]. For end-to-end evaluations, we conduct the inference of typical LLMs on the NVIDIA A100-SXM4-80GB DGX platform with CUDA 11.8. The inference latency and the latency breakdown (Section 7.3) are measured using NVIDIA Nsight System [24].

7.1 Linear Layer Speedups to 8-/16- bit

Workloads. We evaluate the performance of TC-FPx on linear layers under different shapes, coming from the shapes of the weight matrices within LLaMA models [33] (llama-7b, llama-13b, llama-33b, and llama-65b) and OPT models [41] (OPT-30b, OPT-65b, and OPT-175b). We evaluate two versions of our TC-FPx kernel, including the W6A16 (FP6_e3m2) version and the W5A16 (FP5_e2m2) version. For each model, we evaluated the latency of each GPU kernel at three typical inference batch sizes, i.e., 8, 16, and 32.

[10] "eXmY" here means that the model weights are stored in a float-point format with X exponent bits and Y mantissa bits.

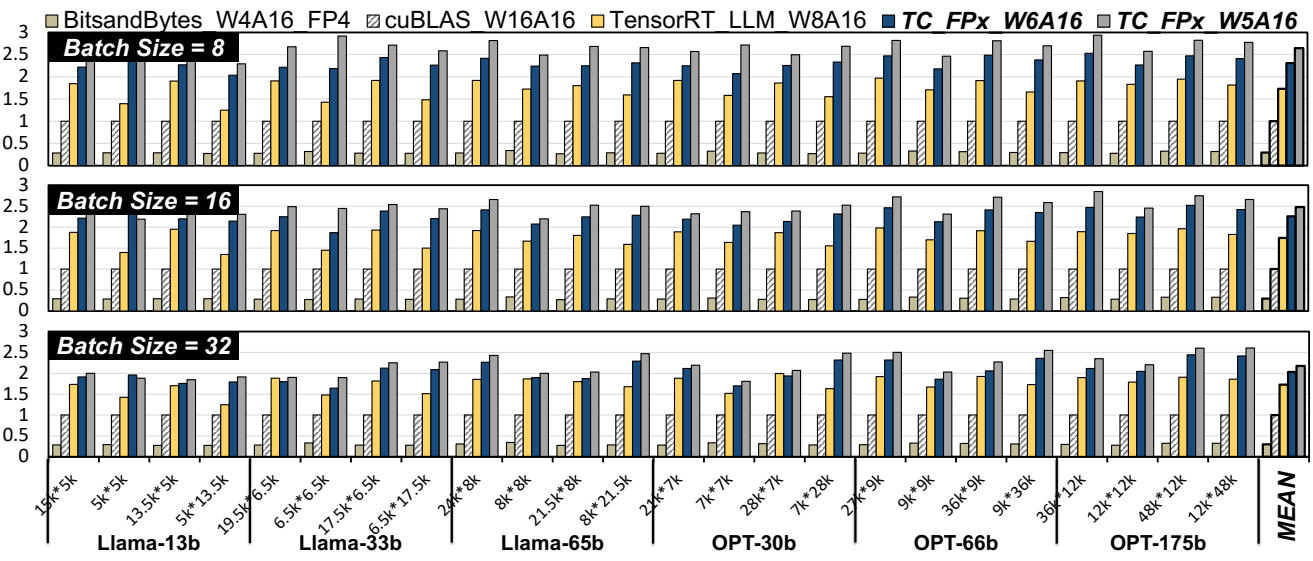


Figure 9: Linear layer speedups compared to the FP16 baseline (cuBLAS) for token generation phase. Two versions of our TC-FPx kernels are evaluated, e.g., W6A16 (FP6_e3m2) and W5A16 (FP5_e2m2). The geometric mean is shown on the right.

Baselines. The baselines we compare include the W16A16 kernels from cuBLAS [22] and the W8A16 kernels from TensorRT-LLM (commit: 6837c81) [26]. What's more, we also include the W4A16 (FP4) support from BitsandBytes (commit: f1ef74f) [3] as a baseline.

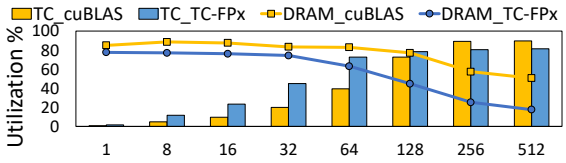
Results. Figure 9 shows the latency speedups of TC-FPx and other baselines. We use the performance of cuBLAS to normalize the performance of all GPU kernels. As shown in Figure 9, TC-FPx_W6A16 outperforms BitsandBytes (W4A16), cuBLAS (W16A16), and TensorRT_LLM (W8A16, INT8 weights) by up to 8.9×, 2.6×, and 1.9×. On average, TC-FPx_W6A16 outperforms BitsandBytes, cuBLAS and TensorRT_LLM by 7.8×/7.5×/6.6×, 2.3×/2.2×/2.0×, and 1.4×/1.3×/1.2× when the batch size is 8/16/32, respectively. Higher speedups can be achieved if the model weights are quantized into FP5. On average, TC-FPx_W5A16 outperforms BitsandBytes, cuBLAS and TensorRT_LLM by 9.0×/8.3×/7.0×, 2.6×/2.4×/2.1×, and 1.6×/1.4×/1.3× when the batch size is 8/16/32, respectively.

Performance Analysis With extensive kernel profiling, We demonstrate the utilization[11] of each GPU hardware unit and provide more in-depth insights into the source of our performance improvements. During the execution of linear layers, as for the cuBLAS baseline, the DRAM bandwidth (shown as the yellow lines in Figure 10a) is almost exhausted (>80%) while the GPU Tensor Cores (shown as the yellow bar in Figure 10a) are not fully used (<50%), when the inference batch size is smaller than 128. It is a common issue during the inference of large language models caused by the **auto-regressive inference** scheme of large language models. With our support of 6-bit quantization, the DRAM access is significantly reduced (up to 2.7×), mitigating the bottleneck of insufficient DRAM bandwidth. Consequently, the Tensor Cores can be more effectively utilized for matrix computations, shown as blue bars compared to yellow bars in Figure 10a. In summary, our kernel mitigated the "memory wall" issue and achieved higher computational efficiency (higher utilization of Tensor Cores) by supporting 6-bit quantization on Tensor Cores.

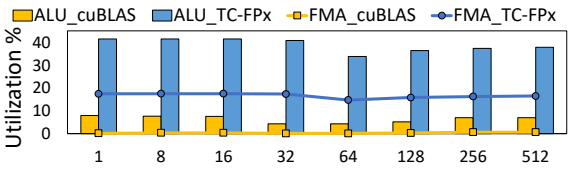
Furthermore, it explains that our kernel can outperform TensorRT-LLM's W8A16 kernel because we are more effective in reducing DRAM access of model weights. Note that the performance of our TC-FPx kernel, cuBLAS kernel, and TensorRT-LLM's W8A16 kernel will eventually converge to the same performance when the inference batch size is larger (bigger than 128), as their performance will all be bounded by the peak computing power of Tensor Cores.

We also observed that BitsandBytes is constantly slower than cuBLAS, which is 29.6% as fast as cuBLAS on average. After further investigation, we found that BitsandBytes adopted the dual-kernel method (discussed in Section 4.1) to support FP4 quantization. During the execution of the first kernel, the FP4 model weights will be first loaded from global memory, de-quantized into FP16, and then written back to global memory in the FP16 data type. After that, a normal cuBLAS kernel is launched computing the matrix multiplication as the second kernel. Thus, the FP4 GPU kernel is always slower than the original FP16 cuBLAS kernel due to the overhead of the extra GPU kernel for FP4 de-quantization.

[11] "Utilization" typically refers to the degree to which a particular hardware resource is being actively used during the execution of a GPU kernel.



(a) Tensor core and DRAM utilization at different batch sizes.



(b) ALU and FMA unit utilization at different batch sizes.

Figure 10: Performance Analysis of TC-FPx_W6A16. SIMT and Tensor Cores are independent computational units whose utilization rate can be measured independently using NVIDIA's profiling tool [23]. As shown in this Figure, "TC_TC-FPx" and "ALU_TC-FPx" are the measured runtime utilization of Tensor Cores and SIMT Cores.

Analysis of on-the-fly De-quantization Figure 10b shows the overhead of FP6-to-FP16 de-quantization in two aspects. On one hand, the FP6-to-FP16 de-quantization introduces a significant number of bit-wise operations even with our SIMT-efficient designs. As a result, the utilization of the Arithmetic/Logic Unit (ALU) has increased from 6.36% to 38.8% on average. It is also strong evidence that the SIMT-efficient designs (Section 5.3) for de-quantization are essential. On the other hand, the FP6-to-FP16 de-quantization also introduces more float-point multiplications, computing the multiplication between the weights and the quantization scales. On average, the utilization of the FMA unit is increased from 0.33% to 16.64%. Given that both ALU and FMA units are part of the SIMT cores, the de-quantization operations will not consume the computing power of Tensor Cores. More importantly, the runtime overhead of SIMT cores can be effectively hidden by overlapping these SIMT instructions with other operations, with our novel designs described in Section 5.4.

7.2 Performance Comparison to 4-bit

Workloads As described in Section 3, 6-bit quantization is more appealing than 4-bit quantization in terms of preserving model quality. However, we still compare the performance of our W6A16 kernels to the state-of-the-art W4A16 kernels, fully demonstrating that our 6-bit quantization can achieve comparable inference speed to the existing 4-bit quantization methods. We evaluate the performance of the linear layers within the LLaMA-65b model [33] under different batch sizes.

Baselines The major baselines here include the W4A16 support of row-wise quantization (Coarse-grained_W4A16) and the W4A16 support of group-wise quantization (Fine-grained_W4A16) from TensorRT-LLM [26] (commit: 6837c81) with state-of-the-art performance. We also include cuBLAS [22] here as the performance baseline, clearly showing the benefits of each quantization method.

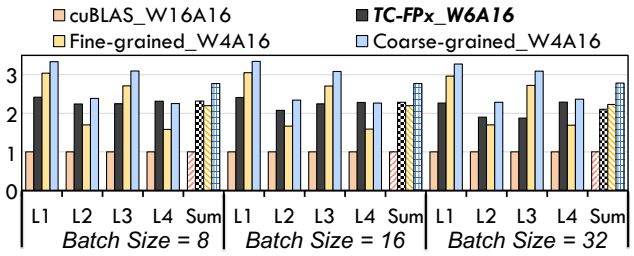


Figure 11: Linear layer speedups compared to using 4-bit weights for token generation phase of the LLaMA-65b model.

Results Figure 11 shows the latency speedups of TC-FPx and other baselines running four different linear layers (i.g. L1, L2, L3, and L4) within the LLaMA-65b models. We use cuBLAS' performance to normalize the performance of other GPU kernels. As shown in Figure 11, TC-FPx_W6A16, Fine-grained_W4A16, and Coarse-grained_W4A16 outperform cuBLAS_W16A16 by up to 2.4×, 3.0×, and 3.3×. More importantly, TC-FPx achieves similar performance with Fine-grained_W4A16, which is 1.06×/1.04×/0.94× faster than Fine-grained_W4A16 when running all these linear layers at batch size 8/16/32, respectively. Besides, TC-FPx is only 16% / 17% / 24% slower than Coarse-grained_W4A16 at batch size 8/16/32. Since 6-bit quantization can provide significantly higher model quality, it is a worthwhile trade-off.

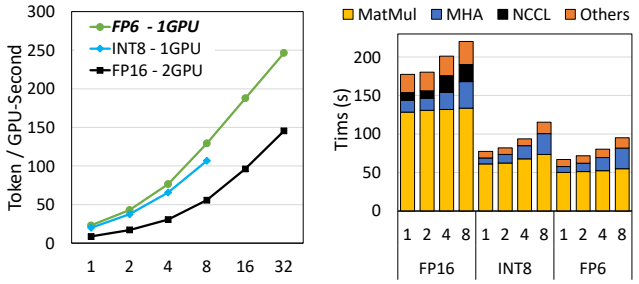
7.3 End2End Inference

Workloads We evaluate the end-to-end inference performance of Quant-LLM on two typical large language models, i.g., LLaMA-70b [34] and OPT-30b [41]. For each model, we evaluated its token generation throughput at different batch sizes, starting from 1 until GPU memory is exhausted.

Metric. We use the metric **tokens per GPU-second** to indicate the *normalized inference throughput* with the consideration of both execution time and hardware cost (i.e., the number of GPUs used). It is calculated with this equation:

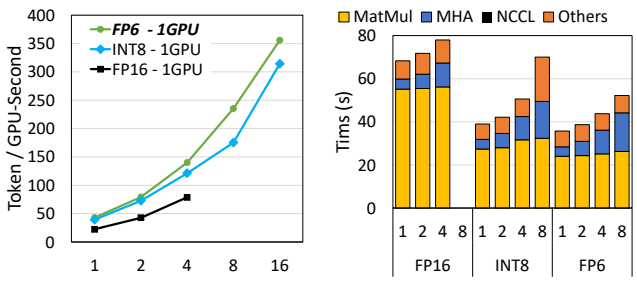
$$Inference\_Performance = \frac{N_{token}}{\sum_{i=1}^{N_{GPU}} T_i} \quad (4)$$

N_{token} means the number of tokens generated, whereas N_{GPU} and T_i mean the GPU number and the time spent on the i'th GPU for execution. We use this metric to evaluate the end-to-end inference performance in this section.



(a) Generation throughput. (b) Inference latency breakdown.

Figure 12: LLaMA-70b inference at different batch sizes. *MatMul*: linear layers, implemented with cuBLAS or our TC-FPx; *MHA*: multi-head attention; *NCCL*: cross-GPU communications; *Others*: other GPU kernels or GPU idle time.



(a) Generation throughput. (b) Inference latency breakdown.

Figure 13: OPT-30b inference at different batch sizes.

Settings and Baselines We set the prefill/prompt length of each request to 0.5K and generate 1.5K tokens for each request, ignoring the *"EOS"* (end of sequence) token. We integrate our TC-FPx kernel into DeepSpeed [19] for end-to-end evaluation and call this new system support Quant-LLM. We use the FP16 execution of the original DeepSpeed system as a baseline for end-to-end comparison. Besides, we integrate the W8A16 kernels of TensorRT-LLM [26] to DeepSpeed and run INT8 inference as another baseline. With our Quant-LLM, only a single 80GB A100 GPU is used for the inference for all the workloads, including the LLaMA-70b model [34]. In contrast, two 80GB A100 GPUs are used for the inference of the LLaMA-70b model for the FP16 baseline, since the model weights ($\approx$ 130 GB) can not be fit into a single GPU.

LLaMA-70b Figure 12a shows token generation throughput on the LLaMA-70b model using our Quant-LLM (FP6-1GPU), compared to the FP16 (FP16-2GPU) and INT8 (INT8-1GPU) baselines. According to our experiments, both our Quant-LLM and FP16 baseline can at most set the inference batch size to 32 before running out of GPU memory, whereas Quant-LLM only requires a single GPU and the FP16 baseline uses two GPUs. The results show that Quant-LLM can achieve 1.69×-2.65× higher normalized inference throughput than the FP16 baseline. Meanwhile, the INT8 baseline can at most set batch size to 8 before running out of GPU memory, while our FP6 solution can set batch size to 32. As a result, our FP6 solution can at most achieve 246 tokens per GPU-second (2.31× higher) with batch size 32, while the INT8 baseline can at most achieve 107 tokens per GPU second with batch size 8, given the same GPU budget. When evaluated with the same batch sizes (1/2/4/8), our FP6 solution can achieve $1.14\times - 1.21\times$ higher throughput than the INT8 baseline.

We conduct a careful latency breakdown of this end-to-end inference process in Figure 12b. Given that two GPUs are used in the FP16 baseline, the execution time on both GPUs is summed in this Figure for fair comparisons. As shown in Figure 12b, our TC-FPx kernel (used in our FP6 Quant-LLM) is 2.40× faster than cuBLAS kernel (used in FP16 baseline) on average. Besides, the NCCL [25] overhead (cross-GPU communications) is fully avoided using Quant-LLM since only a single GPU is required. Overall, our Quant-LLM achieves up to 2.65× higher throughput than the FP16 baseline as for *tokens per GPU-second*. Moreover, our FP6 TC-FPx is $1.22\times - 1.34\times$ faster than the INT8 kernels when setting batch size to 1/2/4/8, resulting in up to 1.21× higher throughput comparing our FP6 solution to the INT8 solution.

OPT-30b Figure 13a shows token generation throughput on the OPT-30b model using our Quant-LLM (FP6-1GPU), compared to the FP16 (FP16-1GPU) and INT8 (INT8-1GPU) baselines. According to our experiments, Quant-LLM can at most set the inference batch size to 16 before running out of GPU memory while the FP16 baseline can at most serve 4 requests in a batch. As a result, Quant-LLM can at most achieve 356 tokens per GPU-second (4.51× higher) with batch size 16 while the FP16 baseline can at most achieve 79 tokens per GPU-second with batch size 4, given the same GPU budget. Besides, Quant-LLM can achieve 1.91×/1.86×/1.78× higher generation throughput than the FP16 baseline when their batch sizes are set to 1/2/4. These overall performance improvements mainly come from the reduction of time in executing linear layers. As shown in Figure 13b, TC-FPx kernel is 2.27× faster than the FP16 cuBLAS kernel on average. Furthermore, our FP6 TC-FPx kernel is $1.14\times - 1.26\times$ faster than the INT8 kernels, resulting in $1.09\times - 1.34\times$ higher end-to-end inference throughput compared to the INT8 solution.

8 Related Work

Six-bit Quantization [36] shows that FP6 performs robustly across various algorithms and tasks, demonstrating its superiority in accuracy and versatility. Besides, [32] verified that the FP6 data format can closely match FP32 for inference after quantization-aware fine-tuning. However, there is no existing hardware support for the proposed data types. Their

inference/training experiments can only be done via software emulations. Quant-LLM can provide high-performance GPU support for the inference of LLMs after FP6 quantization. NVIDIA recently announced FP6 Tensor Cores that would be added to their next generation of GPUs (NVIDIA Blackwell [27]) in the future, indicating that FP6 matters. Compared to this hardware-bounded support of FP6, our software-based techniques (TC-FPx) are generic enough to be applied to a wide range of GPU architecture generations and vendors.

GPU Support for Irregular Bit-widths There is no existing hardware/software support for irregular bit-widths (non-power of 2, e.g., 6-bit, 5-bit) on GPU Tensor Cores. Although supporting irregular bit-width quantization is important for industrial deployment and algorithm exploration, it is extremely challenging on Tensor Cores (See Section 4.2.1). *We are the first to provide complete system designs for unified Tensor Core support of various quantization bit-widths, including irregular bit-widths.* Although it would be easier to support irregular bit-widths only using SIMT Cores (without Tensor Cores), the performance scalability would be poor (See Figure 1). GPTQ [5] supports 3-bit, and Llama.cpp [7] supports 2/3/4/5/6 bit quantization, but they only use SIMT Cores and leave Tensor Cores idle.

GPU Support for Regular Bit-widths Existing Tensor Core based solutions [14,26] only support regular bit-widths (power of 2), e.g., 4-bit and 8-bit. TensorRT-LLM [26] has state-of-the-art kernel supports for weight-only quantization. However, it only supports weights in INT4 (W4A16 [6, 14]) or INT8 (W8A16 and W8A8 [39]) data types while we provide better trade-offs by supporting weights in 6 bits. Besides, TensorRT-LLM does not support float-point data type (e.g., FP6), which is much more complicated to de-quantize during runtime than the integer type. Bitsandbytes [3] mainly supports INT8 weights (W8A8) and has very naive support for FP4 (W4A16) with poor performance. AWQ [14] only has GPU kernel implementation [15] for INT4 (W4A16) in PyTorch. Flash-LLM [37] studies the Tensor Core computation on irregular weight format (i.e., unstructured sparsity), but does not support irregular bit-width and quantization. To the best of our knowledge, our work is the first GPU system that can support FP6 weights on Tensor cores.

9 Conclusions

In this paper, we introduce TC-FPx, the first full-stack GPU kernel design scheme with unified tensor core support for float-point weights of various quantization bit-width, mitigating the "memory wall" issues during LLM inference. We integrate TC-FPx kernel into a state-of-the-art inference system, providing new end-to-end support (called Quant-LLM) for quantized LLM inference, where better trade-offs between inference cost and model quality are achieved. Quant-LLM tackles the problems of hardware-unfriendly memory access and high computation overhead of de-quantization with a set of novel techniques, achieving faster inference speed with significantly less GPU memory. Evaluations show that Quant-LLM enables the inference of LLaMA-70b using only a single GPU, achieving 1.69×-2.65× higher normalized inference throughput than the FP16 baseline. Besides, Quant-LLM improves the inference throughput of OPT-30b by 1.78×-4.51×.

References

[1] 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.

[2] 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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] Tim Dettmers. bitsandbytes. "https://github.com/TimDettmers/bitsandbytes", 2023.

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

[5] 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.

[6] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Optq: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*, 2022.

[7] Georgi Gerganov. llama.cpp. "https://github.com/ggerganov/llama.cpp", 2023.

[8] Github. Copilot. "https://github.com/features/copilot", 2022.

[9] Google. Bard. "https://bard.google.com/", 2023.

[10] Wei Huang, Xudong Ma, Haotong Qin, Xingyu Zheng, Chengtao Lv, Hong Chen, Jie Luo, Xiaojuan Qi, Xianglong Liu, and Michele Magno. How good are low-bit quantized llama3 models? an empirical study, 2024.

[11] William Kahan. Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[12] Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W Mahoney, et al. Full stack optimization of transformer inference: a survey. *arXiv preprint arXiv:2302.14017*, 2023.

[13] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

[14] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration, 2023.

[15] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. llm-awq. "https://github.com/mit-han-lab/llm-awq", 2023.

[16] Bingchang Liu, Chaoyu Chen, Cong Liao, Zi Gong, Huan Wang, Zhichao Lei, Ming Liang, Dajun Chen, Min Shen, Hailian Zhou, Hang Yu, and Jianguo Li. Mftcoder: Boosting code llms with multitask fine-tuning. *arXiv preprint arXiv*, 2023.

[17] Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Using Large Corpora*, page 273, 1994.

[18] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017.

[19] Microsoft. Deepspeed github. "https://github.com/microsoft/DeepSpeed", 2023.

[20] NVIDIA. Nvidia a100 tensor core gpu architecture. "https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf", 2020.

[21] NVIDIA. Nvidia h100 tensor core gpu architecture. "https://www.hpctech.co.jp/catalog/gtc22-whitepaper-hopper_v1.01.pdf", 2022.

[22] NVIDIA. cublas. "https://developer.nvidia.com/cublas", 2023.

[23] NVIDIA. Nsight compute profiling guide. "https://docs.nvidia.com/nsight-compute/ProfilingGuide/#introduction", 2023.

[24] NVIDIA. Nsight system. "https://developer.nvidia.com/nsight-systems", 2023.

[25] NVIDIA. Nvidia collective communications library (nccl). "https://developer.nvidia.com/nccl", 2023.

[26] NVIDIA. Tensorrt-llm. "https://github.com/NVIDIA/TensorRT-LLM/", 2023.

[27] NVIDIA. Nvidia blackwell architecture technical brief. "https://resources.nvidia.com/en-us-blackwell-architecture", 2024.

[28] OpenAI. Chatgpt. "https://openai.com/blog/chatgpt", 2022.

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

[30] Irina Proskurina, Luc Brun, Guillaume Metzler, and Julien Velcin. When quantization affects confidence of large language models?, 2024.

[31] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[32] Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, Stosic Dusan, Venmugil Elango, Maximilian Golub, Alexander Heinecke, Phil James-Roxby,

Dharmesh Jani, Gaurav Kolhe, Martin Langhammer, Ada Li, Levi Melnick, Maral Mesmakhosroshahi, Andres Rodriguez, Michael Schulte, Rasoul Shafipour, Lei Shao, Michael Siu, Pradeep Dubey, Paulius Micikevicius, Maxim Naumov, Colin Verrilli, Ralph Wittig, Doug Burger, and Eric Chung. Microscaling data formats for deep learning, 2023.

[33] 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. Llama: Open and efficient foundation language models, 2023.

[34] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[35] 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.

[36] Xiaoxia Wu, Haojun Xia, Stephen Youn, Zhen Zheng, Shiyang Chen, Arash Bakhtiari, Michael Wyatt, Yuxiong He, Olatunji Ruwase, Leon Song, and Zhewei Yao. Zeroquant(4+2): Redefining llms quantization with a new fp6-centric strategy for diverse generative tasks. *arXiv preprint arXiv: 2312.08583*, 2023.

[37] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity. *Proc. VLDB Endow.*, 17(2):211–224, oct 2023.

[38] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. Flash-llm github. "https://github.com/AlibabaResearch/flash-llm", 2023.

[39] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models, 2023.

[40] Zhewei Yao, Xiaoxia Wu, Cheng Li, Stephen Youn, and Yuxiong He. Zeroquant-v2: Exploring post-training quantization in llms from comprehensive study to low rank compensation. *arXiv preprint arXiv:2303.08302*, 2023.

[41] 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.

[42] 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, 2023.

[43] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. In *KDD*, 2023.

[44] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. A survey on model compression for large language models. *arXiv preprint arXiv:2308.07633*, 2023.