FlashDecoding++Next: High Throughput LLM Inference with Latency and Memory Optimization

Guohao Dai *Member, IEEE*, Ke Hong, Qiuli Mao, Xiuhong Li, Jiaming Xu, Haofeng Huang, Hongtu Xia, Xuefei Ning, Shengen Yan, Yun Liang, *Senior Member, IEEE*, Yu Wang, *Fellow, IEEE*

***Abstract*—As the Large Language Model (LLM) becomes increasingly important in various domains, the performance of LLM inference is crucial to massive LLM applications. However, centering around the computational efficiency and the memory utilization, the following challenges remain unsolved in achieving high-throughput LLM inference: (1) Synchronous partial softmax update. The softmax operation requires a synchronous update operation among each partial softmax result, leading to ∼20% overheads for the attention computation in LLMs. (2) Under-utilized computation of flat GEMM. The shape of matrices performing GEMM in LLM inference tends to be flat, leading to under-utilized computation and 50% performance loss after padding zeros in previous designs (*e.g.*, cuBLAS, CUTLASS, etc.). (3) Memory redundancy caused by activations. Dynamic allocation of activations during inference leads to redundant storage of useless variables, bringing 22% more memory consumption.**

We present *FlashDecoding++Next*, a high-throughput inference engine supporting mainstream LLMs and hardware backends. To tackle the above challenges, *FlashDecoding++Next* creatively proposes: (1) Asynchronous softmax with unified maximum. *FlashDecoding++Next* introduces a unified maximum technique for different partial softmax computations to avoid synchronization. Based on this, a fine-grained pipelining is proposed, leading to 1.18× and 1.14× for the *prefill* and *decode* phases in LLM inference, respectively. (2) Flat GEMM optimization with double buffering. *FlashDecoding++Next* points out that flat GEMMs with different shapes face varied bottlenecks. Then, techniques like double buffering are introduced, resulting in up to 52% speedup for the flat GEMM operation. (3) Buffer reusing and unified memory management. *FlashDecoding++Next* reuses the pre-allocated activation buffers throughout the inference process to remove redundancy. Based on that, we unify the management of different types of storage to further exploit the reusing opportunity. The memory optimization enables up to 1.57× longer sequence to be processed. *FlashDecoding++Next* demonstrates remarkable throughput improvement, delivering up to 68.88× higher throughput compared to the HuggingFace [1] implementation. On average, *FlashDecoding++Next* achieves 1.25× and 1.46× higher throughput compared to vLLM [2] and TensorRT-LLM [3] on mainstream LLMs.

***Index Terms*—Large language model, inference, computation, memory, efficiency.**

I. Introduction

As the Large Language Model (LLM) achieved unprecedented success in various domains [5]–[8], the LLM inference workload is skyrocketing. For example, OpenAI reports that GPT-4 inference with 8K context length costs $0.03 per 1K input tokens and $0.06 per 1K output tokens [9]. Currently, OpenAI has 180.5 million users and receives over 10 million queries per day [10]. Consequently, the cost to operate OpenAI's model like ChatGPT is approximately $7 million per day for the necessary computing hardware [11]. Thus, achieving high throughput in LLM inference is essential for the service providers to cut costs. Many recent works have proposed techniques to optimize the throughput in LLM inference tasks, including DeepSpeed [12], FlexGen [13], vLLM [2], OpenPPL [14], FlashAttention [15], [16], FlashDecoding [17], TensorRT-LLM [3], and etc [18]–[20].

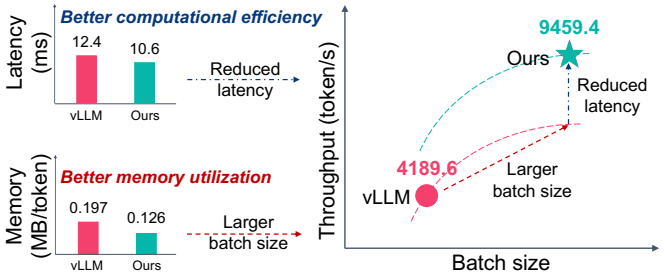


Fig. 1. Overview of comparison between *FlashDecoding++* and the state-of-the-art system. The results in the figure are reported with Llama3-8B model [4]. The results on the left represent the latency with a single batch size and the incremental memory usage per token, and the results on the right show the throughput curves with increasing batch sizes. *FlashDecoding++Next* achieves a significant throughput improvement based on latency and memory optimization. vLLM [2] is the SOTA baseline.

Throughput is defined as the number of output tokens (*e.g.,* some words) that LLM generates per second. Two primary factors influence the throughput of such a system: **computational efficiency** and **memory utilization**. For a given workload, enhanced computational efficiency results in reduced latency per token, thereby increasing throughput. Several previous works, including DeepSpeed [12], FlashAttention [15], [16], FlashDecoding [17], and TensorRT-LLM [3] incorporate highly optimized GPU kernels to accelerate the LLM inference process. Moreover, substantial memory capacity is essential for managing heavy workloads, allowing full exploitation of the hardware's computational capabilities. FlexGen [13] employs the offloading technique to expand the total available memory capacity, while vLLM [2] introduces paged storage to address memory fragmentation, significantly enhancing memory utilization during LLM inference.

The LLM inference task generates tokens from the input

Guohai Dai and Jiaming Xu are with Qing Yuan Research Institute, Shanghai Jiao Tong University, China.

Ke Hong, Qiuli Mao, Haofeng Huang, Xuefei Ning, Shengen Yan, and Yu Wang are with the Department of Electronic Engineering, Tsinghua University, China.

Guohai Dai, Ke Hong, Qiuli Mao, Jiaming Xu, and Hongtu Xia are also with Infinigence-AI.

Xiuhong Li, Hongtu Xia, and Yun Liang are with Peking University, China.

Guohao Dai (daiguohao@sjtu.edu.cn) is the corresponding author.

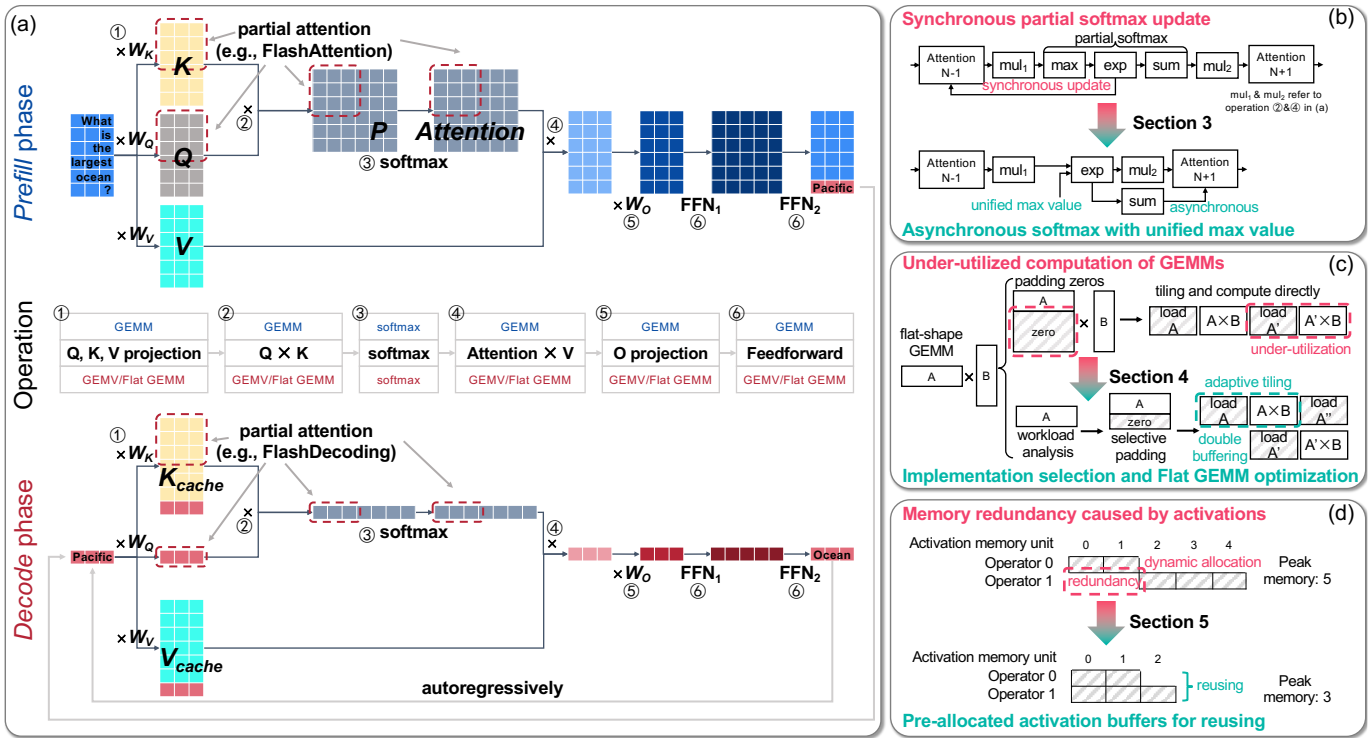


Fig. 2. Overview of Large Language Model inference dataflow. *FlashDecoding++Next* propose three solutions for corresponding challenges in Large Language Model inference. (a) The dataflow comparison between the *prefill* phase and the *decode* phase. The *prefill* phase mainly involves the GEMM operation, while the *decode* phase mainly involves the GEMV/Flat GEMM operation. (b) *FlashDecoding++Next* proposes the asynchronous softmax with a unified max value technique, avoiding synchronous updates to previous partial attention results. (c) *FlashDecoding++Next* optimizes flat GEMM by improving computation utilization. (d) *FlashDecoding++Next* removes memory redundancy by pre-allocating and reusing buffers for activation storage.

sequence autoregressively, and can be organized into two typical phases: the *prefill* phase and the *decode* phase. The *prefill* phase generates the first token by processing the input prompt. The *decode* phase generates the following tokens sequentially. The *prefill* phase dominates total time for scenarios of long-sequence input or generating short outputs [21], [22], while the *decode* phase constitutes a significant portion of the time with chatbot scenarios or processing long output sequences [23]. Fig. 2(a) shows the main dataflow of the LLM inference with one transformer layer for both the *prefill* phase and the *decode* phase. A transformer layer can be divided into GEMM (General Matrix Multiplication) operations (*e.g., K, Q, V, O* weight projection and the feedforward) and the attention/softmax computation. The memory consumption during LLM inference primarily comprises three parts: model weights, activations, and the KVcache. The memory space occupied by the model weights remains constant throughout the inference process. Therefore, this discussion focuses on optimizing the memory usage of the KV cache and activations in LLM inference. Previous studies have highlighted the significance of KVcache storage [2], [18], acknowledging that KVcache occupies a dominant portion of overall memory consumption. However, we observe that inefficient management of activation storage also leads to memory redundancy and fragmentation.

For the attention computation, a softmax operation is adopted for a row in the attention matrix. To improve the parallelism, previous designs [15], [17] divide the attention matrices into smaller tiles and rows are also split to compute partial softmax results. A synchronous softmax operation is adopted to update previous partial softmax results when a new partial softmax result is calculated. Such a **synchrononus partial softmax update** accounts for 18.8% for the attention computation of Llama2-7B inference according to our profiling on NVIDIA Tesla A100 GPU with 1024 input length, resulting in the first challenge for accelerating LLM inference. Secondly, **the computation resources is under-utilized for the flat GEMM operation** during the *decode* phase. Because the *decode* phase sequentially generates tokens, the GEMM operation tends to be flat-shape (even turning into the GEMV, *i.e.,* General Matrix-Vector Multiplication, when the batch size is 1). For the small batch size (*e.g.,* 8), previous designs [24], [25] pad the matrix with zeros to perform GEMMs of larger sizes (*e.g.,* 64), leading to over 50% computation under-utilization. Thirdly, **the throughput of LLM inference suffers from the memory redundancy caused by activations**. Current implementations [2], [3], [12], [18] allocate memory space to store the output activations whenever an operator finishes. Such dynamic allocation for the storage of activations at runtime forces 1.22× more memory space to be detained.

To tackle these challenges and enable a high-throughput Large Language Model (LLM) inference, we present *FlashDecoding++Next* in this paper. Compared to our prior work *FlashDecoding++*, this work incorporates memory optimization techniques, creatively forming **a comprehensive perspective of computational efficiency and memory utilization** to enhance inference throughput. *FlashDecoding++Next* includes the

following contributions:

- **Asynchronous softmax with unified maximum.** *FlashDecoding++Next* leverages a unified maximum for different partial softmax computations. Each partial softmax result can be processed individually without synchronous updates. Such a technique leads to 1.18× and 1.14× speedup for attention computation in the *prefill* phase and *decode* phase, respectively. Moreover, for *decode* phase, the sparse implementation based on the proposed method achieves 1.98× speedup over FlashDecoding on average.
- **Flat GEMM optimization with double buffering.** *FlashDecoding++Next* only pads the matrix size to 8 rather than 64 in previous designs for flat-shaped GEMM to improve computation utilization. We point out that flat GEMMs with different shapes face different bottlenecks, and further improve the kernel performance by up to 52% with techniques like double buffering.
- **Buffer reusing and unified management.** *FlashDecoding++Next* pre-allocates activation buffers and reuses the buffers throughout the LLM inference process, so that no extra memory is allocated for activation storage at runtime. Such a reusing technique reduces the activation memory consumption by 22%, and further allows the unified management for both KVcache and activations, leading to 10% more memory space for KVcache usage. The proposed memory optimization allows processing up to 1.57× longer sequence length.

We conduct extensive experiments to validate the efficiency of *FlashDecoding++Next*. *FlashDecoding++* achieves up to **68.88**× higher throughput on NVIDIA GPUs compared with the HuggingFace [1] implementation. The results show that *FlashDecoding++Next* delivers an average of **1.25**× and **1.46**× throughput improvement compared with vLLM [2] and TensorRT-LLM [3], the state-of-the-art LLM inference engines on various LLMs (*e.g.,* Llama3, Qwen1.5, etc.).

The rest of this paper is organized as follows. Section II introduces preliminaries of LLMs and related works on LLM inference acceleration. Our three techniques designed specifically for the attention computation, the GEMMs and the memory usage are detailed in Section III, IV, and V, respectively. Section VI presents the evaluation results. Related works on LLM inference are introduced in Section VII, and Section VIII concludes the paper.

II. Background

A. LLM Inference Dataflow Overview

The task of LLM inference is to generate tokens from the input sequence, which can be used to complete a sentence or answer a question. An overview of the LLM inference dataflow is shown in Fig. 2(a). As we can see, the LLM inference dataflow can be organized into two typical phases with similar operations: one *prefill* phase and several *decode* phases. The *prefill* phase "understands" the input sequence (*i.e.,* "What is the largest ocean?"). Each token (we set one word as a token in Fig. 2(a)) is encoded as an embedding vector, and the input sequence is organized into a matrix. The main output of the *prefill* phase is a new token, which is predicted to be the next token after the input sequence (*i.e.,* "Pacific" in this figure). The *decode* phase "generates" the output sequence (*i.e.,* "Pacific", "Ocean", etc.) The output token of the *prefill* phase is taken as the input of the *decode* phase. The *decode* phase is executed autoregressively, and each output token is used as the input token for the next *decode* phase (*e.g.,* "Ocean" is further used as the input).

B. Operations in LLM Inference

The main operations in LLM inference are depicted as operation ① to ⑥ in Fig. 2(a), including the linear projection (① and ⑤), the attention (②, ③, and ④), and the feedforward network (⑥). For simplicity, operations like position embedding [26], non-linear activation [27], [28], mask [26], and others are not shown in the figure. Operations in the *prefill* phase and the *decode* phase are different in the shape of data. Because only one token (batch size=1) or few tokens (batch size>1) are processed at one time, **input matrices in the *decode* phase are flat-shape matrices or even vectors.**

Linear Projection. The linear projection performs as the fully connected layer, multiplying the input with weight matrices (*i.e.,* W_K, W_Q, W_V, W_O, called K, Q, V projection and O projection). For the *prefill* phase, the K, Q, V projection generates matrices K, Q, V. For the *decode* phase, the K, Q, V projection generates three corresponding vectors and concatenated with K and V (*i.e.,* KVcache, yellow and light blue in Fig. 2(a)) in the *prefill* phase.

Attention. The attention operation is mainly divided into three operations (② to ④ $Q \times K$, $softmax$, $Attention \times V$), as shown in Eq. (1). For $P = Q \times K^T$, the softmax operation is performed for each row of the result matrix of P. The detailed softmax computation is shown in Fig. 3(a). The maximum value $m(x)$ is first calculated. The exponent of each element divided by $e^{m(x)}$, $f(x)$, is then processed. These exponents are normalized to the summation of all exponents (*i.e.,* $l(x)$) to get the softmax result.

$$softmax(Q \times K^T) \times V \tag{1}$$

Feedforward Network. The feedforward network primarily comprises two fully connected layers. The first one (⑥ FFN_1) expands the feature dimensions to enhance the representative capacity. The second one (⑥ FFN_2) restores the feature dimensions and serves as the output layer.

The main operations in the linear projection and the feedforward network are GEMMs. As shown in Fig. 4, the attention operations and the GEMMs dominate the runtime of LLM inference. Specifically, attention and GEMMs together occupy more than 90% of the latency for 7B/8B LLMs, with the latency portion of attention growing with the input sequence length. Although parallelism brings communication overhead for 70B/72B LLMs, attention and GEMMs still together take up over 70% of the latency. Thus, the key to improving computation efficiency lies in optimizing attention and GEMM operations.

C. Attention Optimization

The softmax operation shown in Fig. 3(a) requires all global data to be calculated and stored before it can proceed. This

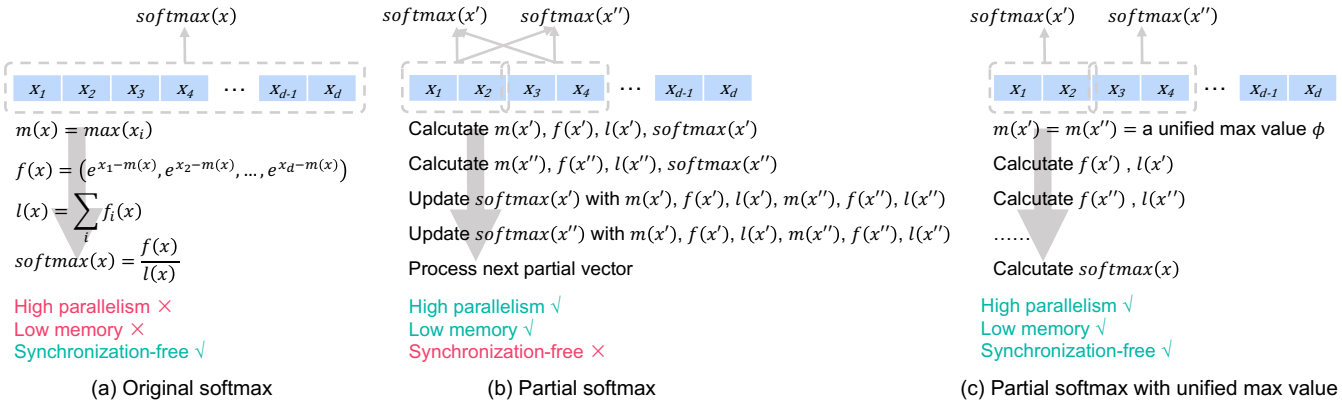


Fig. 3. Comparison of different softmax computation schemes. (a) Softmax computation for the whole vector. (b) Computing partial softmax for each partial vector, and a synchronous update operation is required for all partial softmax results. (c) Computing partial softmax using a unified max value, and each partial vector is processed individually without synchronous updates.

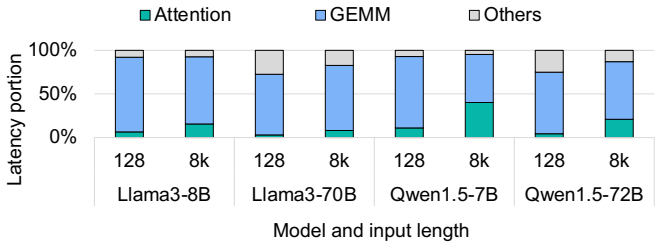


Fig. 4. Inference runtime breakdown across different LLMs and input sequence lengths with vLLM [2]. Attention operations and GEMMs are generally dominant. The output length is set to 128 and the batch size is fixed to 16 for all cases.

results in high memory consumption and low parallelism. Latter works propose the partial softmax technique to reduce memory consumption [15], [16] or improve parallelism [17]. Fig. 3(b) shows the diagram of the partial softmax operation. The main idea is to divide the vector x into partial vectors (*i.e,* x' and x''). The partial softmax results of x' and x'' are calculated separately according to Fig. 3(a), and then synchronously updated by each other. The detailed computation of this synchronized update is shown in Equation (2). With the implementation of partial softmax, we can achieve efficient parallelism of computation while reducing memory cost for attention computation.

$$\begin{aligned} m(x) &= max(m(x'), m(x'')) \\ f(x') &= e^{m(x')-m(x)} f(x') \\ f(x'') &= e^{m(x'')-m(x)} f(x'') \\ l(x) &= f(x') + f(x'') \\ softmax([x', x'']) &= [f(x'), f(x'')] \div l(x) \end{aligned} \tag{2}$$

However, since the partial softmax needs to be updated according to other partial softmax results, it unavoidably introduces data synchronization operations. According to our profiling result, such a synchronized update operation leads to 18.8% overheads in the attention computation on the NVIDIA Tesla A100 GPU with 1024 input length.

D. Tiling in GEMM Optimization

GEMMs can be represented using M, N, K, where the sizes of two multiplied matrices are $M \times K$ and $K \times N$. Tiling is an essential technique for efficiently computing GEMMs on GPUs. In this approach, the original matrices are tiled along all the dimensions into multiple sub-matrices and distributed among different computing units to facilitate parallel processing. Previous works [24], [25] discuss the impact of different tiling strategies in computing GEMMs. The widely-used GEMM library on NVIDIA GPUs, cuBLAS [24], applies several tile sizes (*e.g.*, 64, 128, and 256) to different workloads to maintain versatility. However, the coarse-grained tiling employed by cuBLAS is sub-optimal for computing the flat-shape GEMMs during the *decode* phase.

III. Attention Optimization: Asynchronous Softmax with Unified Maximum

Motivation. The partial softmax operation requires synchronization among different partial vectors, leading to ∼20% overheads of the attention operation. As is shown in Fig. 2(b), synchronization is required after the maximum value of the partial vector is calculated. The maximum value is used to update previous partial softmax (*i.e.,* recompute previous attention) results. Thus, to reduce synchronization overheads, **the key problem to be solved is how to compute each partial softmax result without requiring results from other partial softmax computation.**

Challenge. The reason that synchronization is required lies in that the maximum value of each partial vector is different. The maximum value is used to avoid overflow of the exponent operation ($f(x)$ in Fig. 3(a)), and exponents are summed ($l(x)$ in Fig. 3(a)) as the denominator of the softmax operation. Such a non-linear operation on each partial maximum value makes the synchronization among each partial softmax computation unavoidable.

Analysis and Insights. According to the formula of softmax computation, the maximum value is used as the scaling factor for both the numerator and the denominator (*i.e.,* $f(x)$ and $l(x)$ in Fig. 3(a)). Our key insight is, **the scaling factor can be an arbitrary number** rather than using the maximum value

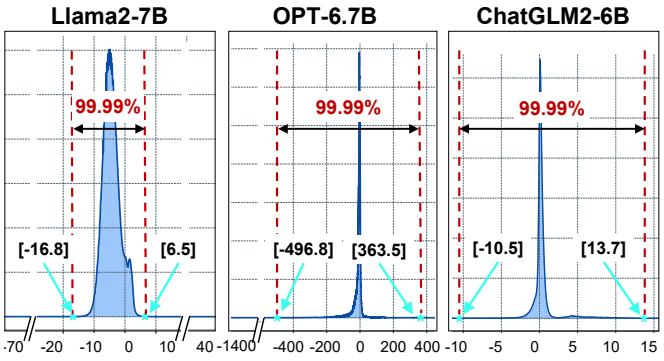


Fig. 5. The statistical distribution of x_i (elements in the input vectors of softmax) in typical LLMs with different inputs.

mathematically, shown in Equation (3). When we set $\phi = 0$, it becomes the original softmax computation [29].

$$softmax(x) = \frac{[e^{x_1 - m(x)}, ..., e^{x_d - m(x)}]}{\sum_i e^{x_i - m(x)}} = \frac{[e^{x_1 - \phi}, ..., e^{x_d - \phi}]}{\sum_i e^{x_i - \phi}}, \forall \phi \in \mathbb{R} \quad (3)$$

However, the scaling factor cannot be an arbitrary number considering the overflow of the exponent computation. For the case where $x_i \gg \phi$, $e^{x_i - \phi}$ overflows and cannot be represented using a fixed-width floating point number (*e.g.,* `float32` for exponent results in current LLM engines). For another case where $x_i \ll \phi$, $e^{x_i - \phi} \to 0$, leading to precision loss. Thus, a proper scaling factor ϕ should be carefully selected to avoid the two cases above. Fig. 5 shows the statistical distribution of x_i (elements in the input vectors of softmax) in typical LLMs with different inputs [30]. Our key insight is, $> 99.99\%$ x_i **are within a certain range**. Specifically, for Llama2-7B, we have $-16.8 < x_i < 6.5$ for $> 99.99\%$ x_i. Because e^{b-a} and e^{a-b} can be represented by a `float32` format, we can set $\phi = a$ in Equation (3). For OPT-6.7B, we do not apply the technique in this section because of the large range in Fig. 5.

Approach: Asynchronization. Based on the insights above, each partial softmax computation shares a unified maximum value ϕ. After the softmax operation, an inner product operation is executed between the softmax result and a column of V (*i.e.,* v). Assume that the input vector x can be divided into p partial vectors, $x = [x^{(1)}, ..., x^{(p)}]$ ($v = [v^{(1)}, ..., v^{(p)}]$ correspondingly), we have:

$$\langle softmax(x), v \rangle = \frac{\sum_i e^{x_i - \phi} \cdot v_i}{\sum_i e^{x_i - \phi}} = \frac{\sum_{j=1}^{p} \sum_{i=1}^{d/p} e^{x_i^{(j)} - \phi} \cdot v_i^{(j)}}{\sum_{j=1}^{p} \sum_{i=1}^{d/p} e^{x_i^{(j)} - \phi}} \quad (4)$$

The inner accumulation in both the numerator and the denominator only takes the partial vectors $x^{(j)}$ and $v^{(j)}$ as input, thus they can be processed asynchronously and individually. The outer accumulation is only processed after all partial vectors are processed. As we can see in Fig. 3(c), each $f(x^{(j)})$ is calculated individually, and $softmax(x)$ is calculated after all $x^{(j)}$ is calculated.

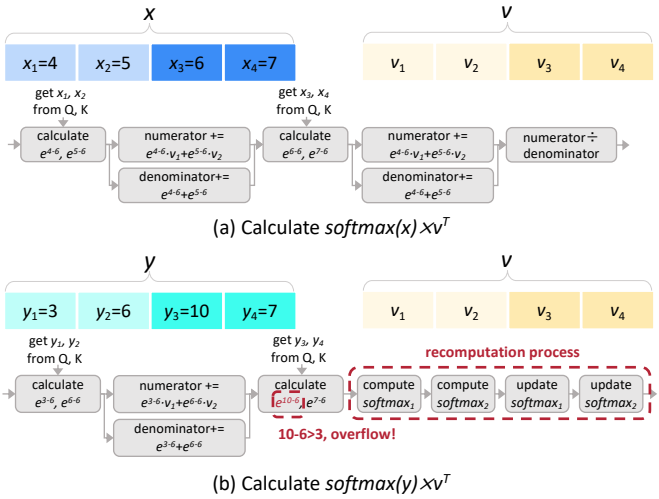


Fig. 6. Example of asynchronous partial softmax computation. (a) Each partial softmax result is processed individually without the synchronous update. (b) The recomputation process for all partial softmax computation is required when an overflow happens.

Approach: Recomputation. Without loss of generality, we assume $a < x_i - \phi < b$ for each x_i to ensure precision and avoid overflow. Then, the partial softmax operation is processed individually. However, when $x_i - \phi \leq a$ or $x_i - \phi \geq b$, the asynchronous partial softmax computation is terminated for the vector x where x_i belongs to. The softmax is then recomputed using the synchronized partial softmax scheme (used in FlashAttention [15], [16] and FlashDecoding [17]) shown in Fig. 3(b). Such a recomputation scheme avoids overflow while introducing negligible overheads based on the statistical data shown in Fig. 5.

Example. Fig. 6 shows an example of the asynchronous softmax scheme. We set $a = -3, b = 3, \phi = 6$. Two vectors x and y are calculated from $Q \times K^T$ in Equation (1), and are divided into 2 partial vectors. We omit the process from $Q \times K^T$ to these partial vectors. For each x_i, we have $a < x_i - \phi < b$, we process $e^{x_1 - \phi} \cdot v_1 + e^{x_2 - \phi} \cdot v_2$ and $e^{x_1 - \phi} + e^{x_2 - \phi}$ for the first partial vector of x using two asynchronous threads. Then, each thread moves to the next partial vector for the corresponding computation (*i.e.,* $e^{x_3 - \phi} \cdot v_3 + e^{x_4 - \phi} \cdot v_4$ and $e^{x_3 - \phi} + e^{x_4 - \phi}$). Two threads are synchronized when all partial vectors are processed, and perform the division operation in Equation (4). For y, the first partial vector is processed similarly. However, we find that $y_3 - \phi > b$, then two threads are terminated and the first thread recomputes all partial vectors according to the synchronous partial softmax scheme in Fig. 3(b).

Sparse Attention Support. For the *decode* phase, *FlashDecoding++Next* supports the sparse attention pattern in MoA [31], which selectively sparsifies the KVcache for each head. To adapt to the varying KVcache sizes across different attention heads, *FlashDecoding++Next* achieves workload balance by reordering the computation of attention heads. We follow the longest-processing-time-first (LPT) rule for GPU streaming multiprocessor (SM) scheduling. Specifically, we permute the Q, K, V, O projection weights offline to handle attention head reordering, and prioritize the heads with longer

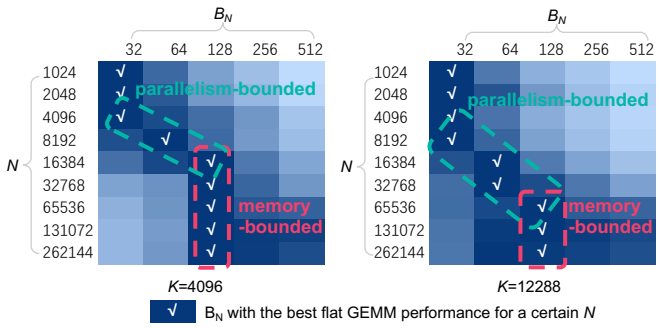


Fig. 7. Normalized flat GEMM performance under different $N-$dimension sizes and $N-$dimension tiling sizes. We set $M = 8$ and execute GEMM on the NVIDIA Tesla A100 GPU.

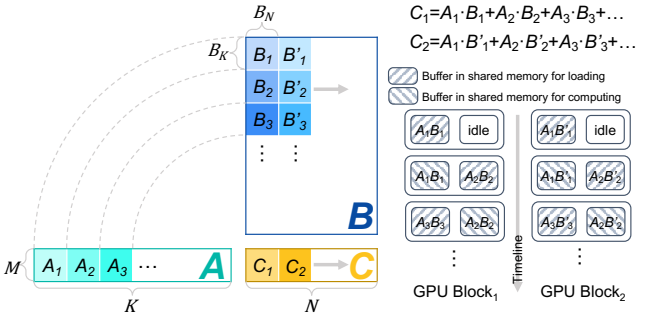


Fig. 8. Double buffering for flat GEMM when $N-$dimension is large. The $M-$ dimension is padded to 8 and not tiled.

KVcache lengths. Such prioritization ensures minimal idle time for the SMs, thereby minimizing the total execution time of the attention operator.

IV. GEMM Optimization: Flat GEMM with Double Buffering

Motivation. The process of the *decode* phase is mainly composed of GEMV (batch size=1) or flat GEMM (batch size>1) operations. Without loss of generality, GEMV/GEMM operations can be represented using M, N, K, where the sizes of two multiplied matrices are $M \times K$ and $K \times N$. Previous LLM inference engines utilize Tensor Core to accelerate these operations using libraries like cuBLAS [24] and CUTLASS [25]. Although modern Tensor Core architectures [32] process GEMM with $M = 8$, these libraries usually tile the $M-$dimension to 64 to hide memory latency. However, for GEMV or flat GEMM operations in the *decode* phase, we may have $M \ll 64$, and the $M-$dimension is padded to 64 with zeros. The padding leads to under-utilized computation, and **the key problem is to process GEMV or flat GEMM operations with smaller tiles (*i.e.,* padding to 8 corresponding to modern Tensor Core architectures) in the $M-$dimension**.

Challenge. Processing GEMV or flat GEMM operations is non-trivial when the $M-$dimension is padded to 8. The tiling technique in modern libraries like cuBLAS [24] and CUTLASS [25] can only be applied to the $N-$dimension and the $K-$dimension. Tiles on the $K-$dimension are processed sequentially in a GPU block to avoid atomic operations during reduction. Tiling on the $N-$dimension affects both parallelism and computation/memory ratio, which are both important for GEMV and flat GEMM acceleration.

Analysis and Insights. Assume that tiling sizes of the $N-$dimension and the $K-$dimension are B_N and B_K, respectively. The computation of each GEMM tile is $2\times M\times B_N\times B_K$ with total $B = \frac{N\times K}{B_N\times B_K}$ GEMM tiles. The total memory access is $(M \times B_K + B_N \times B_K) \times B + M \times N$. Thus, the computation/memory ratio is:

$$\frac{2 \times M \times B_N \times B_K \times B}{(M \times B_K + B_N \times B_K) \times B + M \times N} = \frac{2 \times M \times K}{K + \frac{M\times K}{B_N} + M} \tag{5}$$

On the other hand, the parallelism is $\frac{N}{B_N}$. Thus, the computation/memory ratio shows a positive correlation with B_N while the parallelism shows a negative correlation with B_N, exposing a contradiction in improving the performance of GEMV or flat GEMM. We depict the normalized performance of the flat GEMM in Fig. 7 with different N and B_N. Our key insight is, **for the smaller N, the flat GEMM is parallelism-bounded**. There are 108 Streaming Multiprocessors (SMs) in the NVIDIA Tesla A100. $\frac{N}{B_N}$ tends to be a constant (*e.g.,* 128 or 256), which is related to the hardware parallelism (number of SMs). Another key insight is, **for the larger N, the flat GEMM becomes memory-bounded**. The performance of these cases can be improved by hiding memory access latency.

Approach: Tile Size and Double Buffering. We adopt small tile sizes (*e.g.*, 32 or 64) with small N to generate more tiles for parallel computing. Considering the memory access bottleneck with large N, we utilize the double buffering technique to hide memory access latency. Specifically, we allocate two separate buffers in the shared memory. The tile in one buffer performs the GEMM operation, while another buffer loads a new tile for the next GEMM operation. Thus, the computation and the memory access are overlapped. We apply such a technique when N is large in our practice.

Approach: Implementation Selection. Notably, the application of Tensor Core is not necessary for Flat GEMM implementation, which inevitably leads to memory access and computation redundancy when M (*i.e.*, the batch size) is extremely small. Such workload can be optimized by utilizing CUDA Core in previous designs like FastGEMV [33]. For a Llama3-8B linear layer in the *decode* phase, the Tensor Core implementation only achieves 82.15% of the performance of CUDA Core implementation on an NVIDIA A100 GPU. Thus, we form a selection algorithm to decide whether to use Tensor Core for the linear workloads in LLM inference.

Although influential factors including input dynamics and hardware capacities form a large search space, the homogeneity of different layers in LLM significantly reduces the search space for implementation selection. Fig. 2(a) shows four linear operations in the *prefill* phase and the *decode* phase, *i.e.,* K, Q, V projection, O projection, and two feedforward operations. Each GEMM operation can be abstracted as a multiplication between an $(M \times K)$-shaped matrix and a $(K \times N)$-shaped matrix, and **there are only four $[K, N]$ shapes for a certain LLM**. We profile the performance of the

implementations mentioned above for a certain M, and increase M to find an inflection point $\hat{M}$ where the performance of using Tensor Core is better than using pure CUDA Core. For the runtime LLM inference, *FlashDecoding++Next* adopts the implementation using CUDA Core when $M < \hat{M}$, and the Tensor Core implementation when $M \geq \hat{M}$. Note that the decision flow is executed offline, it does not affect the performance of runtime LLM inference.

Example. Fig. 8 shows the example of our flat GEMM optimization with double buffering. For $M < 8$, the $M-$dimension is first padded to 8 considering modern Tensor Core architectures. Workloads in the $K-$dimension are processed within one GPU block (*e.g.,* $A_1, A_2, A_3, ...$), while workloads in the $N-$dimension are processed in parallel using different GPU blocks (*e.g.,* $C_1, C_2, ...$). We take GPU Block$_1$ as an example, the first tile for each matrix in the $K-$dimension (*i.e.,* A_1 and B_1) is loaded to the left buffer in the shared memory. Then, the GEMM operation is performed between A_1 and B_1. Consequently, A_2 and B_2 are loaded to the right buffer in the shared memory. The following tiles are processed similarly according to the double buffering scheme.

V. Memory Optimization: Buffer Reusing and Unified Management

Motivation. As mentioned in Section I, the storage in LLM inference consists of the model weights, the KVcache, and the activations. The memory space occupied by the model weights remains constant throughout the inference process. Thus, we discuss the KVcache and the activations for memory optimization in LLM inference. Each transformer layer contains multiple operators, such as attention, GEMMs, and element-wise operators. The input and output tensors for those operators, *i.e.*, activations, require memory allocation that cannot be immediately released after use. As depicted in Fig. 9(a), dynamic allocation ignores the lifecycle of those activations at runtime, and **allocates new memory space to store the output of each operator, unnecessarily resulting in a 22% higher peak memory usage**. Since the memory allocated for KVcache and activations cannot be reused between each other, dynamic allocation of activation space also reduces the available space for KVcache storage. As illustrated in Fig. 10(a), activation memory usage is more consumed during the *prefill* phase, whereas KV cache memory usage peaks during the *decode* phase. **This discrepancy between the storage demands between the *prefill* and *decode* phases results in a 10% redundancy in previous approaches.** Therefore, it is essential to reuse the same memory space for different storage to eliminate memory redundancy.

Challenge. In the original implementation, the code logic is typically organized in the unit of a single transformer layer. The input and output tensors of operators (*i.e.*, activations) within the same layer span the entire transformer layer, thus necessitating different memory addresses for storage. Considering data dependencies and tensor size variations, it is challenging to further partition the layer-wise activation lifespan into smaller ones. Furthermore, current systems typically pre-allocate the memory space for KVcache while dynamically allocating activations at runtime, making it challenging to reuse the memory space between KVcache and activations.

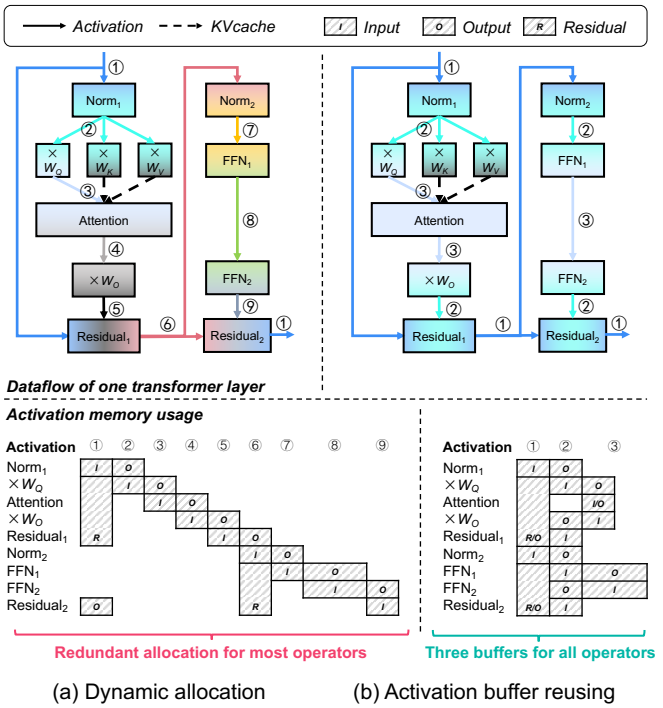


Fig. 9. Memory allocation and usage of activations. (a) Dynamic allocation allocates a new memory space to store activation whenever an operator finishes. (b) Only three activation buffers are needed via buffer reusing.

Analysis and Insights. We observe that the architectures of mainstream transformer-based LLMs are highly similar. Differences include whether the model has biases in W_q, W_k, W_v (*e.g.*, Qwen series does while Llama series does not) and the use of Grouped Query Attention (GQA), which leads to a smaller KVcache. Additionally, the dataflow among different transformer layers within the same model remains entirely consistent. Those observations indicate that **the memory space for activations can be pre-determined and pre-allocated before runtime**. Pre-determined activation storage facilitates the precise allocation of memory between the KV cache and activations, enabling unified memory management for both the *prefill* and *decode* phases. Furthermore, the pre-allocation manner eliminates the overhead associated with multiple memory allocation operations.

Approach: Buffer Reusing. Based on the above insight, we designed a memory-reusing scheme for all activations within a single layer. Element-wise operations such as residual connections and SiLU [28] functions can utilize the same memory address for the input and output activations. For other operations such as normalizations and GEMMs, the input and output activations must use different memory addresses to avoid write-after-read (WAR) conflicts. Considering that the activation used for residual is maintained throughout the processing of the entire layer, at least three memory buffers are needed for the activations within a layer, and the three memory buffers are further reused among different layers. As shown in Fig. 9(b), we reuse three memory buffers to store all the activations within one layer. The three activation buffers have the shapes

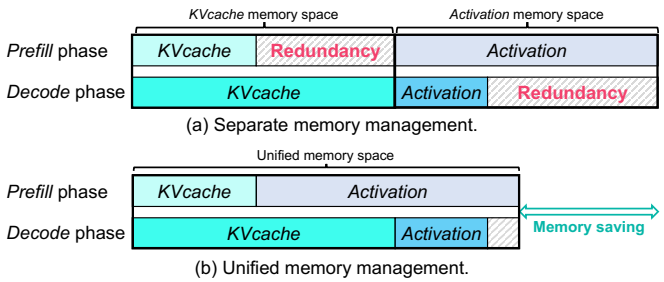


Fig. 10. Memory management of KVcache and activations. (a) Separate management brings redundancy in both KVcache and activation storage. (b) Unified management eliminates redundancy and saves memory space.

of $[S, H \times D], [S, H \times D], [S, F \times H \times D/T]$, respectively. Note that S denotes the total input sequence length of all the requests, H and D represent the head number and head size in the attention operation. F is the multiplier of the FFN hidden size (*e.g.*, 3.5 for Llama3 series), and T is the degree of tensor parallelism. Specifically, we store the input, the output of the attention operator, and the output of FFN_1 in buffer ③ (shape: $[S, F \times H \times D/T]$), as they are all affected by tensor parallelism. Such a storage pattern ensures no wastage under different tensor parallelism settings. Moreover, we store the input of FFN_1 and the output of FFN_2 in buffer ② (shape: $[S, H \times D]$). Buffer ① (shape: $[S, H \times D]$) stores the other activations, which have the same size as the hidden state, *i.e.*, the input of the layer.

Approach: Unified Management. Based on the pre-allocated activation buffers, we further apply a unified memory space to manage the activations and KVcache together. As shown in Fig. 10(a), the application of a separate memory space brings redundancy to the KVcache memory in the *prefill* phase and to the activation memory in the *decode* phase. We propose to switch a portion of the activation memory in the *prefill* phase to store the KVcache in the *decode* phase, which is illustrated in Fig. 10(b). Such a reusing scheme necessitates the unification of the data layout as well. Since the storage of activations necessitates contiguous memory along the second dimension, we take the value $H \times D/T$ as the minimal storage unit for unified management. The shape of buffer ③ is correspondingly adjusted to $[S, \lceil F \rceil \times H \times D/T]$. Furthermore, we ensure the last two dimensions for KVcache storage are H and D, so the minimal storage unit still holds for managing KVcache. Notably, to facilitate unified memory management, the KV cache uses ascending address allocation, while activations use descending address allocation. Such a bidirectional strategy prevents memory conflicts between the two types of storage before the memory space reaches full capacity.

Example. Fig. 9 shows the example of the proposed buffer reusing scheme within one transformer layer. There are two types of storage in the figure, where the dotted line represents KVcache and the solid line denotes activations. We reuse three buffers to store all the activations. Buffer ① is assigned for all residual activations. Buffer ② stores the input activations of the query projection ($\times W_q$) and FFN_1, as well as the output activations of the output projection ($\times W_o$) and FFN_2. Buffer ③ is utilized to store the output activations of the query projection and FFN_1, along with the input activations of the output projection and FFN_2. The output activation of the query projection and the input activation of the output projection are stored in the two segments of the buffer ③ without data conflict.

TABLE I
MODEL CONFIGURATION

Model	Hidden Dimension	FFN Dimension	Attention	Tensor Parallelism
Llama3-8B	4096	14336	GQA	TP=1
Llama3-70B	8192	28672	GQA	TP=4
Qwen1.5-7B	4096	11008	MHA	TP=1
Qwen1.5-72B	8192	24576	MHA	TP=4

VI. EVALUATION

A. Experiments Setup

We evaluate the performance of *FlashDecoding++Next* on different GPUs with various Large Language Models. We compare the performance with several state-of-the-art LLM inference engines.

1) Hardware Platforms: We conduct experiments on a server with 8 pairwise connected NVIDIA A100 80GB SXM4 GPUs and Intel Xeon Platinum 8358P CPU @ 2.60GHz, and the corresponding software environment includes CUDA 12.1 [34], PyTorch 2.3.1 [35]. We also use the NVIDIA RTX 3090 24GB GPU, the AMD MI210 GPU, and the AMD RX7900XTX GPU to test the adaptability.

2) LLM Engine Baselines: We implement our *FlashDecoding++Next* using the Pytorch-based front-end with the C++ and CUDA backend for NVIDIA GPUs. We compare the inference performance with the following LLM engine baselines: HuggingFace (HF) v4.43.2 [1], vLLM v0.5.3 [2], TensorRT-LLM v0.12.0 [3].

- **HuggingFace** is a community that supports flexible development for diverse LLMs. To achieve that, its inference engine (transformers [1]) follows a highly modular design at the cost of relatively sub-optimal inference performance.
- **vLLM** is one of the state-of-the-art inference engines widely employed by researchers and service providers, which is characterized by its efficient GPU kernel for attention operation and memory management optimization for KVcache.
- **TensorRT-LLM**, the official LLM inference engine by NVIDIA, contains highly optimized kernels tailored for NVIDIA GPUs, and hence delivers outstanding inference performance.

3) Models: We evaluate the performance of *FlashDecoding++Next* with other LLM inference engines on two typical LLM series: Llama3 series [4] and Qwen1.5 series [36]. Since different types of attention affect both computation and memory, we choose LLMs that utilize GQA (Llama3 series) and MHA (Qwen1.5 series), respectively. Table 2 shows the detailed configuration of these models. Note that there may be several models for one LLM series (*e.g.,* Llama3-8B, Llama3-70B for Llama3 series) with different configurations (*e.g.,* number of heads and layers).

4) Metrics: To evaluate the end-to-end performance, we choose the highest achievable throughput as the main metric. The highest achievable throughput reflects both the computation efficiency and the memory utilization of the system, and can directly transfer to the cost. Moreover, we compare the speedup and the memory usage under the same batch size to compare the computation efficiency and the memory utilization, respectively.

B. Comparison with State-of-the-art

We compare *FlashDecoding++Next* with state-of-the-art LLM inference engines in Fig. 11. The highest achievable throughput of each inference engine is recorded. *FlashDecoding++Next* outperforms all the baselines across different models and different input lengths. On average, the highest achievable throughput of *FlashDecoding++Next* is **1.25**× and **1.46**× higher compared to vLLM and TensorRT-LLM, respectively. Notably, *FlashDecoding++Next* delivers up to **68.88**× higher throughput than HF, with Llama3-70B and 16k input length. TensorRT-LLM delivers a comparable performance as vLLM and *FlashDecoding++Next* with Llama3 series but becomes sub-optimal with Qwen1.5 series. We observe that the enhancement of *FlashDecoding++Next* becomes significant with larger models, as the increased memory requirement during the generation process amplifies the effectiveness of the proposed memory optimizations.

To comprehensively understand the advantage of *FlashDecoding++Next*, we show the detailed throughput numbers under different batch sizes. As illustrated in Fig. 12, *FlashDecoding++Next* and TensorRT-LLM achieve similar throughput under the same batch size, but TensorRT-LLM tends to run out of memory (OOM) with large batch sizes, thereby falling short in obtaining higher throughput. *FlashDecoding++Next* delivers a better computational efficiency than vLLM, leading to a higher throughput under the same batch size. And *FlashDecoding++Next* also shows a superior memory utilization than vLLM, as vLLM runs OOM earlier with the Qwen1.5 series (lower throughput caused by preemption with Qwen1.5-7B and no space for activations with Qwen1.5-72B). The speedup of *FlashDecoding++Next* under the same batch size benefits from the proposed operator-level optimizations, which become diminished on larger models due to the increased communication overhead across GPUs.

We further compare the throughput of *FlashDecoding++Next* with baselines on AMD GPUs to demonstrate its adaptability, as depicted in Fig. 13 and Fig. 14. The benchmarking models include Llama2-7B, Llama2-13B [37], and OPT-6.7B [38]. Note that the evaluated version of vLLM does not support AMD RX7900XTXA GPUs. *FlashDecoding++Next* achieves up to 2.41× and 4.35× higher throughput compared with Hugging Face on a single RX7900XTX and MI210 GPU, respectively. Notably, the average speedup of *FlashDecoding++Next* compared to vLLM is 1.86×.

C. Ablation Studies

1) End-to-End Throughput Breakdown: As shown in Fig. VI-C1, we detail the throughputs of the implementation

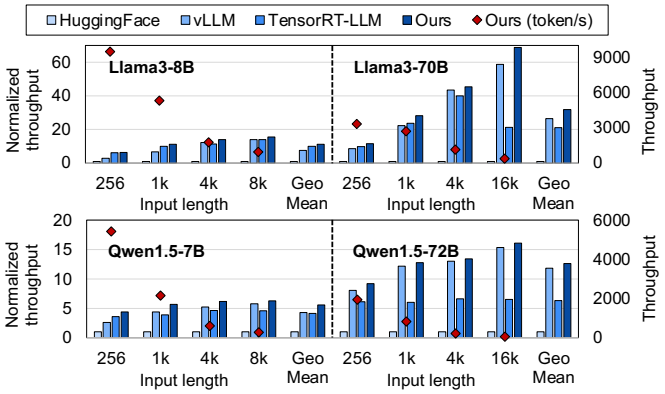


Fig. 11. The highest achievable throughput comparison. The throughput of each engine is normalized to the throughput of the HF implementation. And the absolute throughput of *FlashDecoding++Next* is shown as "◊".

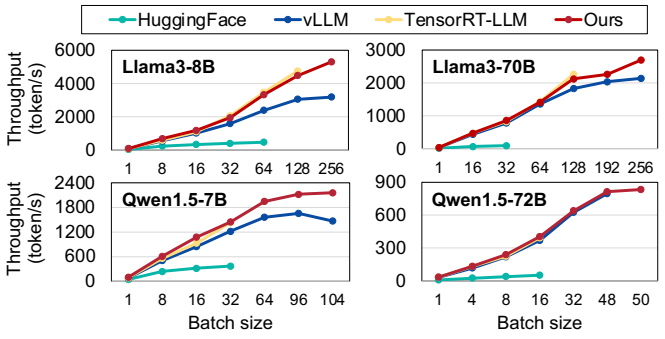


Fig. 12. The throughput comparison under the same batch size. Data left blank denotes the situation of running out of memory.

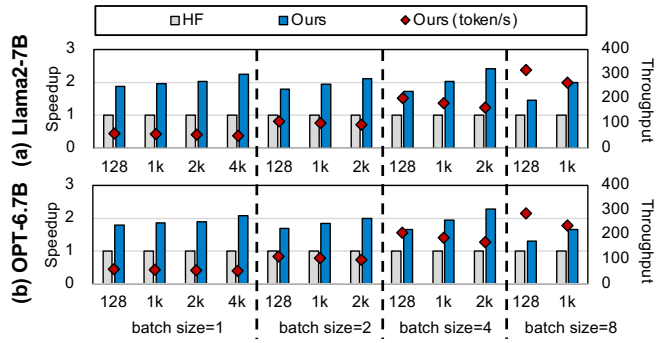


Fig. 13. Speedup of the *decode* phase on AMD RX7900XTX.

without the memory optimization (w/o Memory in the Table) and the implementation without both the memory optimization and the flat GEMM optimization (w/o Memory, Flat GEMM in the table). Notably, the memory optimization enables *FlashDecoding++Next* to execute under 2× larger batch sizes, leading to 1.11× and 1.14× higher achievable throughputs on Qwen1.5-7B and Llama3-70B, respectively. The flat GEMM optimization mainly benefits the inference under small batch sizes, and improves the throughputs by up to 1.11×. Given a fixed batch size, the application of the memory optimization brings different impacts on different models. Specifically, the memory optimization necessitates customized kernels for FFN computation, which leads to either performance degradation (*e.g.*, Llama3-70B) or enhancement (*e.g.*, Qwen1.5-7B) for certain shapes.

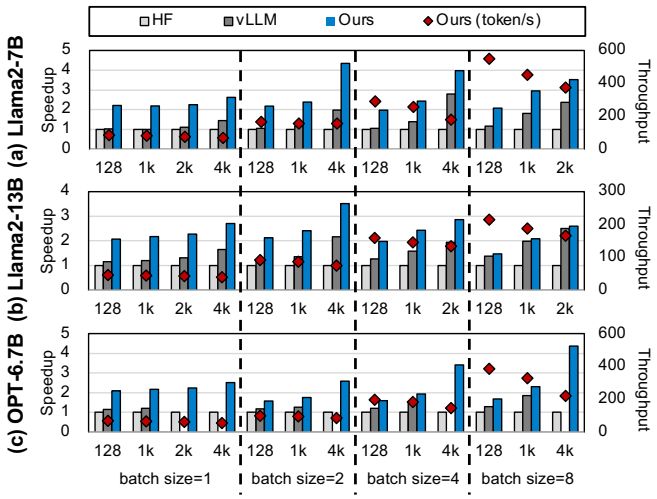


Fig. 14. Speedup of the *decode* phase on AMD MI210. There are blank bars for vLLM because it doesn't support sequence length over 2k for OPT-6.7B.

TABLE II
THROUGHPUT BREAKDOWN

Batch size	1	8	16	32	64	104
Qwen1.5-7B						
Full	100.2	609.3	1076.2	1446.4	1946.6	2158.0
w/o Memory	99.6	592.47	985.3	1388.0	1811.1	OOM
w/o Memory, Flat GEMM	92.5	582.6	969.6	1389.7	1812.2	OOM
Batch size	1	16	32	64	128	256
Llama3-70B						
Full	35.2	466.4	856.1	1409.0	2123.8	2693.0
w/o Memory	34.0	464.8	895.4	1557.6	2355.5	OOM
w/o Memory, Flat GEMM	30.2	455.4	888.8	1558.1	2361.4	OOM

2) Asynchronous Softmax Computation:

- **Benefits.** The asynchronous softmax scheme can be applied to both the *prefill* phase and the *decode* phase. We test the proposed scheme against the state-of-the-art attention implementations in Fig. 15 and Fig. 16 on NVIDIA GPUs. For the *prefill* phase, *FlashDecoding++Next* achieves 1.52× and 1.19× average speedup compared with xformers [39] and FlashAttetion2. For the *decode* phase, *FlashDecoding++* outperforms the decoding-tailored implementation of xformers (denoted as xformers-decoder in Fig. 16) with short KVcache length, and achieves up to 2.02× speedup over FlashDecoding with long KVcache length. We also evaluate the sparse implementation of *FlashDecoding++Next* for the *decode* phase. We adopt the MoA [31] setting that features a 50% average reduction of the KVcache length in *decode* attention computation. As shown in Fig. 17, the sparse implementation of *FlashDecoding++Next* achieves 1.98× speedup over FlashDecoding on average.
- **Correctness.** The absolute difference between the proposed attention method and PyTorch is average 99.7% < 1e-2, and all < 1e-1 (FlashAttention leads to 99.8% < 1e-2 v.s. PyTorch). As mentioned in Sec. III, we introduce a recomputation mechanism into the asynchronous softmax, which automatically selects FlashAttention for computation when the intermediate results overflow. The frequency of recomputation is statistically

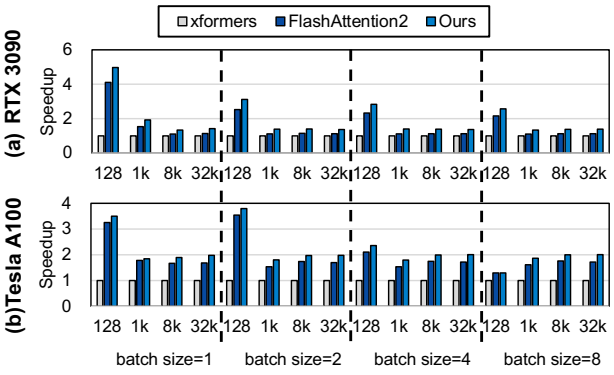


Fig. 15. Benefits of asynchronous softmax (*prefill* phase).

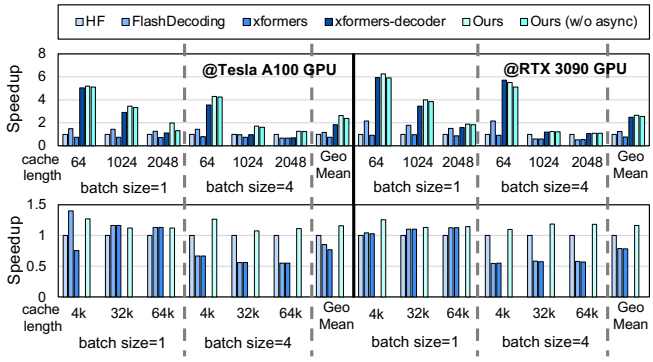


Fig. 16. Benefits of asynchronous softmax (*decode* phase).

obtained to be 0.45% on average across datasets including ARC [40], HellaSwag [41] and Winogrande [42]. We also compare the end-to-end generation results to verify the feasibility of the asynchronous softmax technique. Specifically, we collect the output token indices generated from HuggingFace, vLLM, and *FlashDecoding++Next*, and then compare the indices to those generated by the official Llama implementation [43] on the ShareGPT dataset. Experiments show that for 91.36%, 92.31%, and 91.55% of the input prompts, HuggingFace, vLLM, and *FlashDecoding++Next* generate exactly the same output tokens as the official Llama implementation, respectively. Besides, for the mismatched cases, *FlashDecoding++Next* also exhibits a similar distribution with HuggingFace and vLLM regarding the total different tokens.

- **Adaptability.** We extend our approach to models including CodeLlama-7B [44] and Vicuna-7B [45], which are fine-tuned on Llama2-7B [37] to be applied in specific domains. For both models, the inputs to the softmax operation are obtained through multiple datasets. 99% of the softmax input in CodeLlama-7B ranges from -0.25 to 17.6, while that of Vicuna-7B ranges from -0.8 to 9.8. Thus, the asynchronous softmax method is also applicable to those fine-tuned models. The distribution is significantly correlated with the model structure. Consequently, applying the technique to novel structured models necessitates recollection and analysis of the softmax input data distribution.

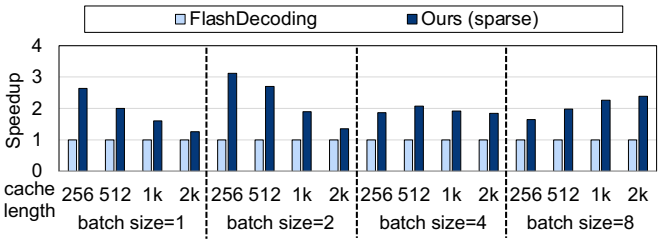


Fig. 17. Sparse attention implementation speedup of *FlashDecoding++Next* (*decode* phase).

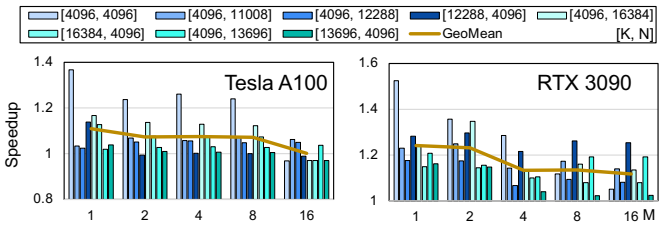


Fig. 18. Speedup over cuBLAS with flat GEMM optimization.

3) Flat GEMM Optimization:

- **Benefits.** We test our flat GEMM kernel performance with the state-of-the-art GEMM library, cuBLAS on two NVIDIA GPUs. The version of cuBLAS is CUDA 11.8. We vary M from 1 to 16 to demonstrate the flat GEMM operation in LLM inference, and eight $[K, N]$ configurations used in typical LLMs are depicted in Figure 18. The flat GEMM optimization in *FlashDecoding++Next* achieves an average of 7% and 17% (up to 52%) speedup on Tesla A100 and RTX 3090, respectively. Libraries including cuBLAS are designed for general purposes, hence not the best for the flat GEMM practice. The speedup is 9% and 23% for small M (*i.e.,* 1 and 2), showing that the proposed flat GEMM optimization explores the computation capability with small batch sizes.
- **Adaptability.**The usage of double buffering with large size in N- dimension is limited by the shared memory (L1 cache) size of GPUs. The results in Figure 18 demonstrate that the strategy works with both NVIDIA Tesla A100 GPUs (192KB L1 cache per SM) and NVIDIA RTX 3090 GPUs (128KB L1 cache per SM) thanks to the large L1 data cache. But for AMD GPUs, double buffering fails to benefit the flat GEMM performance due to a limited L1 data cache (16KB per CU for AMD MI210). Without double buffering, the flat GEMM optimization performs badly in many cases. Note that on AMD GPUs, flat GEMM still benefits from the implementation selection method. The average speedups are 57% and 37% on AMD MI210 GPUs and AMD RX7900XTX GPUs, respectively.

4) Buffer Reusing and Unified Management:

- **Benefits.** We compare the memory usage before and after applying the proposed buffer reusing and unified management optimizations. The results in Fig. 19 demonstrate the efficacy of the optimizations, with an average of 9.75% memory saving. Specifically, for Llama3-8B, the proposed techniques save 17.18% memory usage on average. The reason is that the application of GQA brings a larger portion of memory to activations. The proposed memory optimization improves the longest achievable context length from 320k to 502k (1.57×), and from 110k to 128k (1.16×) for Llama3-8B and Qwen1.5-7B, respectively.
- **Adaptability.** The buffer reusing and unified management technique can be adapted to most mainstream LLMs, but for the mixture-of-experts (MoE) models, the reusing scheme necessitates adjustments considering that each expert holds a separate activation. Therefore, the number and size of buffers, along with the data dependency analysis, need to be redesigned based on the number of experts.

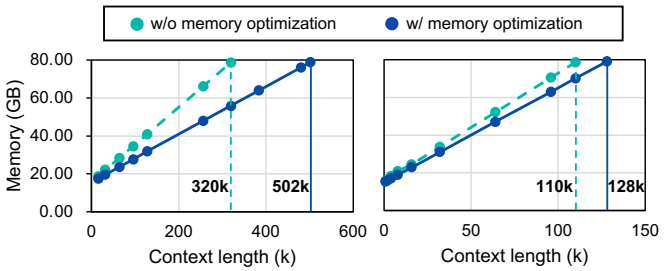


Fig. 19. The memory usage comparison on a single A100 80GB GPU. *Left*: Llama3-8B. *Right*: Qwen1.5-7B.

VII. Related Works

Large language model inference acceleration has gained significant attention in recent research, with several notable approaches and techniques emerging in the field. **DeepSpeed** [12] is a comprehensive engine that optimizes both the training and inference phases for LLMs. It achieves robust inference performance through kernel fusion and efficient GPU memory management, with a particular focus on optimizing memory usage for KVcache. **vLLM** [2] improves GPU memory utilization by efficient memory management techniques and the PageAttention method, leading to increased maximum batch sizes and elevating the upper limit of inference performance. **FlashAttention** [15], [16] optimizes the self-attention computation process during the prefill phase through improved parallelism and workload distribution. **FlashDecoding** [17] is an extension of FlashAttention and enhances the parallelism through splitting K and V, supporting efficient self-attention computation for long sequences during the decode phase. **FasterTransformer** [46] implements large model inference engines using *C++* to reduce overhead resulting from kernels scheduling, compared to *Python* implementations. Those works also employ memory management techniques and kernel fusion to achieve efficient LLM inference. **TensorRT-LLM** [3] is built upon the *TensorRT* [47] and the *FasterTransformer* [46] engine (*C++*) and incorporates cutting-edge open-source technologies such as *FlashAttention* [15], [16]. Additionally, it enhances its ease of use by providing the *Python API*.

VIII. Conclusion

We propose *FlashDecoding++Next*, a high-throughput Large Language Model inference engine in this paper. Centering around computational efficiency and memory utilization, *FlashDecoding++Next* contains three novel designs: the asynchronous softmax with unified maximum, the flat GEMM optimization with double buffering, and the memory optimization based on buffer reusing and unified management. *FlashDecoding++Next* achieves up to **68.88**× higher throughput compared to the HuggingFace implementations. *FlashDecoding++Next* also delivers an average of **1.25**× and **1.46**× throughput improvement compared to the state-of-the-art LLM inference engines, on various LLMs.

References

[1] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, "Transformers: State-of-the-Art Natural Language Processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://aclanthology.org/2020.emnlp-demos.6

[2] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[3] N. Vaidya, F. Oh, and N. Comly, "Optimizing Inference on Large Language Models with NVIDIA TensorRT-LLM, Now Publicly Available," [Online], 2023, https://github.com/NVIDIA/TensorRT-LLM.

[4] Meta, "Introducing Meta Llama 3: The Most Capable Openly Available LLM to Date," April 2024. [Online]. Available: https://ai.meta.com/blog/meta-llama-3/

[5] A. J. Thirunavukarasu, D. S. J. Ting, K. Elangovan, L. Gutierrez, T. F. Tan, and D. S. W. Ting, "Large Language Models in Medicine," *Nature medicine*, vol. 29, no. 8, pp. 1930–1940, 2023.

[6] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, S. Shakeri, E. Taropa, P. Bailey, Z. Chen, E. Chu, J. H. Clark, L. E. Shafey, Y. Huang, K. Meier-Hellstern, G. Mishra, E. Moreira, M. Omernick, K. Robinson, S. Ruder, Y. Tay, K. Xiao, Y. Xu, Y. Zhang, G. H. Abrego, J. Ahn, J. Austin, P. Barham, J. Botha, J. Bradbury, S. Brahma, K. Brooks, M. Catasta, Y. Cheng, C. Cherry, C. A. Choquette-Choo, A. Chowdhery, C. Crepy, S. Dave, M. Dehghani, S. Dev, J. Devlin, M. Díaz, N. Du, E. Dyer, V. Feinberg, F. Feng, V. Fienber, M. Freitag, X. Garcia, S. Gehrmann, L. Gonzalez, G. Gur-Ari, S. Hand, H. Hashemi, L. Hou, J. Howland, A. Hu, J. Hui, J. Hurwitz, M. Isard, A. Ittycheriah, M. Jagielski, W. Jia, K. Kenealy, M. Krikun, S. Kudugunta, C. Lan, K. Lee, B. Lee, E. Li, M. Li, W. Li, Y. Li, J. Li, H. Lim, H. Lin, Z. Liu, F. Liu, M. Maggioni, A. Mahendru, J. Maynez, V. Misra, M. Moussalem, Z. Nado, J. Nham, E. Ni, A. Nystrom, A. Parrish, M. Pellat, M. Polacek, A. Polozov, R. Pope, S. Qiao, E. Reif, B. Richter, P. Riley, A. C. Ros, A. Roy, B. Saeta, R. Samuel, R. Shelby, A. Slone, D. Smilkov, D. R. So, D. Sohn, S. Tokumine, D. Valter, V. Vasudevan, K. Vodrahalli, X. Wang, P. Wang, Z. Wang, T. Wang, J. Wieting, Y. Wu, K. Xu, Y. Xu, L. Xue, P. Yin, J. Yu, Q. Zhang, S. Zheng, C. Zheng, W. Zhou, D. Zhou, S. Petrov, and Y. Wu, "PaLM 2 Technical Report," 2023.

[7] J. Clusmann, F. R. Kolbinger, H. S. Muti, Z. I. Carrero, J.-N. Eckardt, N. G. Laleh, C. M. L. Löffler, S.-C. Schwarzkopf, M. Unger, G. P. Veldhuizen *et al.*, "The Future Landscape of Large Language Models in Medicine," *Communications Medicine*, vol. 3, no. 1, p. 141, 2023.

[8] C. Cui, Y. Ma, X. Cao, W. Ye, and Z. Wang, "Receive, Reason, and React: Drive as You Say with Large Language Models in Autonomous Vehicles," *arXiv preprint arXiv:2310.08034*, 2023.

[9] OpenAI, "OpenAI Pricing," [Online], 2023, https://openai.com/pricing.

[10] Nerdynav, "Up-to-Date ChatGPT Statistics and User Numbers," [Online], 2023, https://nerdynav.com/chatgpt-statistics.

[11] A. A. DYLAN PATEL, "The Inference Cost Of Search Disruption - Large Language Model Cost Analysis," [Online], 2023, https://www.semianalysis.com/p/the-inference-cost-of-search-disruption.

[12] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley *et al.*, "DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–15.

[13] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Re, I. Stoica, and C. Zhang, "FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU," 2023.

[14] Sensetime, "OpenPPL: A High-performance Deep Learning Inference Platform," [Online], 2023, https://openppl.ai/home.

[15] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

[16] T. Dao, "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning," *arXiv preprint arXiv:2307.08691*, 2023.

[17] T. Dao, D. Haziza, F. Massa, and G. Sizov, "Flash-Decoding for Long-Context Inference," [Online], 2023, https://crfm.stanford.edu/2023/10/12/flashdecoding.html.

[18] Sensetime, "A Light and Fast Inference Service for LLM," [Online], 2023, https://github.com/ModelTC/lightllm.

[19] "Text Generation Inference: Fast Inference Optimize for LLMs," [Online], 2023, https://github.com/huggingface/text-generation-inference/.

[20] "MLC LLM: Machine Learning Compilation for Large Language Models," [Online], 2023, https://github.com/mlc-ai/mlc-llm.

[21] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-xl: Attentive Language Models Beyond a Fixed-Length Context," *arXiv preprint arXiv:1901.02860*, 2019.

[22] Z. Dong, T. Tang, L. Li, and W. Zhao, "A Survey on Long Text Modeling with Transformers," *arXiv preprint arXiv:2302.14502*.

[23] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis, "Efficient Streaming Language Models with Attention Sinks," *arXiv preprint arXiv:2309.17453*, 2023.

[24] NVIDIA, "cuBLAS: Basic Linear Algebra on NVIDIA GPUs," [Online], 2017, https://developer.nvidia.com/cublas.

[25] NVIDIA, "CUTLASS: Cuda templates for linear algebra subroutines," [Online], 2017, https://github.com/NVIDIA/cutlass.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is All You Need," *Advances in neural information processing systems*, vol. 30, 2017.

[27] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[28] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for Activation Functions," *arXiv preprint arXiv:1710.05941*, 2017.

[29] J. Bridle, "Training Stochastic Model Recognition Algorithms as Networks Can Lead to Maximum Mutual Information Estimation of Parameters," *Advances in neural information processing systems*, vol. 2, 1989.

[30] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer Sentinel Mixture Models," 2016.

[31] T. Fu, H. Huang, X. Ning, G. Zhang, B. Chen, T. Wu, H. Wang, Z. Huang, S. Li, S. Yan, G. Dai, H. Yang, and Y. Wang, "Moa: Mixture of sparse attention for automatic large language model compression," 2024. [Online]. Available: https://arxiv.org/abs/2406.14909

[32] NVIDIA, "NVIDIA Tensor Core," [Online], 2023, https://www.nvidia.com/en-us/data-center/tensor-cores/.

[33] S. Wang, "FastGEMV: High-Speed GEMV Kernels," [Online], 2023, https://github.com/wangsiping97/FastGEMV.

[34] NVIDIA, "CUDA Toolkit," [Online], June 2024, https://developer.nvidia.com/cuda-toolkit.

[35] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An Imperative Style, High-Performance Deep Learning Library," *Advances in neural information processing systems*, vol. 32, 2019.

[36] A. Yang, B. Yang, B. Hui, B. Zheng, B. Yu, C. Zhou, C. Li, C. Li, D. Liu, F. Huang, G. Dong, H. Wei, H. Lin, J. Tang, J. Wang, J. Yang, J. Tu, J. Zhang, J. Ma, J. Yang, J. Xu, J. Zhou, J. Bai, J. He, J. Lin, K. Dang, K. Lu, K. Chen, K. Yang, M. Li, M. Xue, N. Ni, P. Zhang, P. Wang, R. Peng, R. Men, R. Gao, R. Lin, S. Wang, S. Bai, S. Tan, T. Zhu, T. Li, T. Liu, W. Ge, X. Deng, X. Zhou, X. Ren, X. Zhang, X. Wei, X. Ren, X. Liu, Y. Fan, Y. Yao, Y. Zhang, Y. Wan, Y. Chu, Y. Liu, Z. Cui, Z. Zhang, Z. Guo, and Z. Fan, "Qwen2 Technical Report," *arXiv preprint arXiv:2407.10671*, 2024.

[37] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama

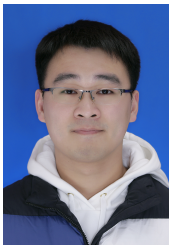
2: Open Foundation and Fine-Tuned Chat Models," *arXiv preprint arXiv:2307.09288*, 2023.
[38] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," 2022.
[39] B. Lefaudeux, F. Massa, D. Liskovich, W. Xiong, V. Caggiano, S. Naren, M. Xu, J. Hu, M. Tintore, S. Zhang, P. Labatut, and D. Haziza, "xFormers: A Modular and Hackable Transformer Modelling Library," https://github.com/facebookresearch/xformers, 2022.
[40] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord, "Think You Have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge," *ArXiv*, vol. abs/1803.05457, 2018.
[41] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi, "HellaSwag: Can a Machine Really Finish Your Sentence?" in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.
[42] K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi, "WinoGrande: An Adversarial Winograd Schema Challenge at Scale," *arXiv preprint arXiv:1907.10641*, 2019.
[43] meta llama, "Inference code for llama models," [Online], https://github.com/meta-llama/llama.
[44] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code Llama: Open Foundation Models for Code," *arXiv preprint arXiv:2308.12950*, 2023.
[45] W.-L. Chiang, Z. Li, Z. Lin, Y. Sheng, Z. Wu, H. Zhang, L. Zheng, S. Zhuang, Y. Zhuang, J. E. Gonzalez, I. Stoica, and E. P. Xing, "Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality," March 2023. [Online]. Available: https://lmsys.org/blog/2023-03-30-vicuna/
[46] NVIDIA, "FasterTransformer: About Transformer related optimization, including BERT, GPT," [Online], 2017, https://github.com/NVIDIA/FasterTransformer.
[47] NVIDIA, "NVIDIA TensorRT: An SDK for High-Performance Deep Learning Inference," [Online], https://developer.nvidia.com/tensorrt.

Guohao Dai (Member, IEEE) received the BS and Ph.D. (with honor) degrees from Tsinghua University, Beijing, in 2014 and 2019. He is currently an associate professor with the Department of Electronic Information and Electrical Engineering at Shanghai Jiao Tong University. His research mainly focuses on large-scale sparse graph computing, heterogeneous hardware computing, emerging hardware architecture, etc. He served as Co-Chair for the Ph.D. Forum at DAC'24, TPC member for ASP-DAC'25/DAC'25/DAC'24/DAC'23/VLSID'24, and reviewer for IEEE TCAD. He has received Best Paper Award in ASP-DAC'19 and DATE'24, and Best Paper Nominations in DATE'23, DAC'22, DATE'18. He is the winner of the NeurIPS Billion-Scale Approximate Nearest Neighbor Search Challenge in 2021, the recipient of the Outstanding PhD Dissertation Award of Tsinghua University in 2019.

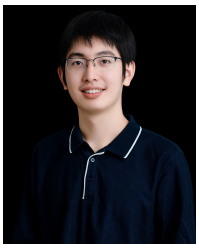
Ke Hong received her BS and Master degrees in electronic engineering from Tsinghua University, Beijing, in 2021 and 2024. She is currently pursuing her Ph.D. degree in Electronics and Communication Engineering at Tsinghua University. Her research interests include machine learning systems and high-performance computing on GPUs.

Qiuli Mao received his BS and Master degrees in electronic engineering from Tsinghua University, Beijing, in 2021 and 2024. He is currently a development engineer at Infinigence-AI. His research interests include deep learning frameworks and high-performance computing on GPUs.

Xiuhong Li received his BS and PhD degrees from Center for Energy-efficient Computing and Applications (CECA), Peking University, China. He is currently a research assistant professor at Peking University. His research interests include high-performance computing on GPUs and deep learning infrastructure.



Jiaming Xu is a Ph.D. student supervised by Prof. Guohao Dai in Shanghai Jiao Tong University. Previously, he obtained his Bachelor's degree from Xidian University in 2023. He is focusing on the system optimization on LLM and Graph. He has published papers on top conferences of both high-performance computing and computer architecture (*e.g.* MLSys and ICCAD).



Haofeng Huang is currently an undergraduate student at the Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing. His academic focus is on algorithm-system co-design aimed at accelerating machine learning workloads.

Hongtu Xia received his B.S. degree in microelectronics from Peking University, Beijing, in 2023. He is currently pursuing his M.S. degree in Electronic and Information Engineering at Peking University. His research interests include computing-in-memory and spiking neural networks.

Xuefei Ning received her B.S. and Ph.D. degrees from Tsinghua University, in 2016 and 2021. She is currently a research-track assistant professor with the Department of Electronic Engineering, Tsinghua University. Her research mainly focuses on efficient deep learning.

Shengen Yan is a Research Associate Professor with the Department of Electronic Engineering, Tsinghua University, Beijing, China. He received the Ph.D degree from Institute of Software, Chinese Academy of Sciences in 2014. He was an Executive Research Director with SenseTime Research from 2015 to 2022. His current research interests include heterogeneous computing, intelligent computing and machine learning systems. Prof. Yan has authored over 40 scientific publications in premier international journals and conferences in related domains.

Yun Liang (Senior Member, IEEE) received the Ph.D. degree in computer science from National University of Singapore, Singapore, in 2010. He is a Professor (with tenure) with the School of EECS, Peking University, Beijing, China. His research interests include computer architecture, compiler, electronic design automation, and embedded system. He has authored over 90 scientific publications in premier international journals and conferences in related domains. Dr. Liang's research has been recognized by Best Paper Awards at FCCM 2011 and ICCAD 2017 and Best Paper Nominations at PPoPP 2019, DAC 2017, ASPDAC 2016, DAC 2012, FPT 2011, and CODES+ISSS 2008. He serves as an Associate Editor for ACM Transactions in Embedded Computing Systems, ACM Transactions on Reconfigurable Technology and Systems, and Embedded System Letters. He also serves for the program committees in the premier conferences in the related domain, including MICRO, DAC, HPCA, FPGA, ICCAD, FCCM, and ICS.

Yu Wang (M'07-SM'14-F'22) received the B.S. and Ph.D. (with honor) degrees from Tsinghua University, Beijing, in 2002 and 2007. He is currently a tenured professor with the Department of Electronic Engineering, Tsinghua University. His research interests include brain inspired computing, application specific hardware computing, parallel circuit analysis, and power/reliability aware system design methodology. He has authored and coauthored more than 300 papers in refereed journals and conferences. He has received the Best Paper Award in ASPDAC 2019, FPGA 2017, NVMSA 2017, ISVLSI 2012, and Best Poster Award in HEART 2012 with 10 Best Paper Nominations. He is a recipient of DAC under 40 innovator award (2018), IBM X10 Faculty Award (2010). He served as TPC chair for ICFPT 2019 and 2011, ISVLSI2018, finance chair of ISLPED 2012-2016, track chair for DATE 2017-2019 and GLSVLSI 2018, and served as program committee member for leading conferences in these areas, including top EDA conferences such as DAC, DATE, ICCAD, ASP-DAC, and top FPGA conferences such as FPGA and FPT. He served as co-editor-in-chief of the ACM SIGDA E-Newsletter, associate editor of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, the IEEE Transactions on Circuits and Systems for Video Technology, ACM Transactions on Embedded Computing Systems, ACM Transactions on Design Automation of Electronic Systems, IEEE Embedded Systems Letters, the Journal of Circuits, Systems, and Computers, and Special Issue editor of the Microelectronics Journal. He is now with ACM SIGDA EC and DAC 2021 EC. He is the co-founder of Deephi Tech (acquired by Xilinx in 2018), which is a leading deep learning computing platform provider.