FLASHDECODING++: FASTER LARGE LANGUAGE MODEL INFERENCE WITH ASYNCHRONIZATION, FLAT GEMM OPTIMIZATION, AND HEURISTICS

Ke Hong [*1,2] **Guohao Dai** [*3,2] **Jiaming Xu** [*3,2] **Qiuli Mao** [1,2] **Xiuhong Li** [4] **Jun Liu** [3,2] **Kangdi Chen** [2] **Yuhan Dong** [1] **Yu Wang** [1]

ABSTRACT

As the Large Language Model (LLM) becomes increasingly important in various domains, the performance of LLM inference is crucial to massive LLM applications. However, the following challenges still remain unsolved in accelerating LLM inference: (1) Synchronized partial softmax update. The softmax operation requires a synchronized update operation among each partial softmax result, leading to ∼20% overheads for the attention computation in LLMs. (2) Under-utilized computation of flat GEMM. The shape of matrices performing GEMM in LLM inference is flat, leading to under-utilized computation and 50% performance loss after padding zeros in previous designs (*e.g.,* cuBLAS, CUTLASS, etc.). (3) Performance loss due to static dataflow. Kernel performance in LLM depends on varied input data features, hardware configurations, etc. A single and static dataflow may lead to a 50.25% performance loss for GEMMs of different shapes in LLM inference.

We present *FlashDecoding++*, a fast LLM inference engine supporting mainstream LLMs and hardware back-ends. To tackle the above challenges, *FlashDecoding++* creatively proposes: **(1) Asynchronized softmax with unified max value.** *FlashDecoding++* introduces a unified max value technique for different partial softmax computations to avoid synchronization. Based on this, the fine-grained pipelining is proposed, leading to 1.18× and 1.14× for the prefill and decoding stage in LLM inference, respectively. **(2) Flat GEMM optimization with double buffering.** *FlashDecoding++* points out that flat GEMMs with different shapes face varied bottlenecks. Then, techniques like double buffering are introduced, resulting in up to 52% speedup for the flat GEMM operation. **(3) Heuristic dataflow with hardware resource adaptation.** *FlashDecoding++* heuristically optimizes dataflow using different hardware resource (*e.g.,* Tensor Core or CUDA core) considering input dynamics. The design leads to up to 29% speedup compared with the static dataflow. Due to the versatility of optimizations in *FlashDecoding++*, *FlashDecoding++* can achieve up to **4.86**× and **4.35**× speedup on both NVIDIA and AMD GPUs compared to Hugging Face implementations. *FlashDecoding++* also achieves an average speedup of **1.37**× compared to state-of-the-art LLM inference engines on mainstream LLMs.

1 INTRODUCTION

As the Large Language Model (LLM) achieved unprecedented success in various domains (Thirunavukarasu et al., 2023; Anil et al., 2023; Clusmann et al., 2023; Cui et al., 2023), 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 (OpenAI, 2023). Currently, OpenAI has 180.5 million users and receives over 10 million queries per day (Nerdynav, 2023). Consequently, the cost to operate OpenAI's model like ChatGPT is approximately $7

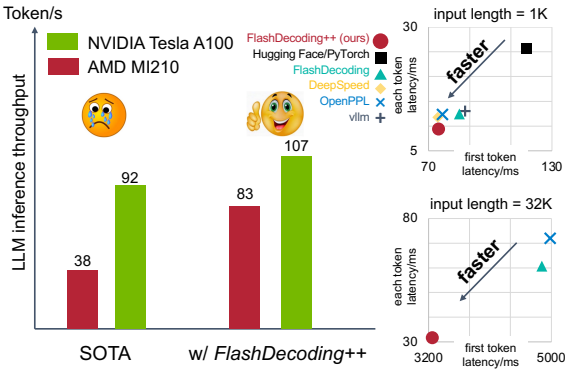


Figure 1. Overview of comparison between *FlashDecoding++* and state-of-the-art designs. The results in the figure are reported with Llama2-7B model (Touvron et al., 2023). The left is with batch size=1 and input length=1K, and TensorRT-LLM and Hugging Face are the SOTA baseline for NVIDIA/AMD according to our experimental results. The right shows the comprehensive comparison of both first token latency and each token latency.

[*]Equal contribution [1]Tsinghua University [2]Infinigence-AI [3]Shanghai Jiao Tong University [4]Peking University. Correspondence to: Guohao Dai <daiguohao@sjtu.edu.cn>, Yu Wang <yuwang@tsinghua.edu.cn>.

Proceedings of the 7th MLSys Conference, Santa Clara, CA, USA, 2024.

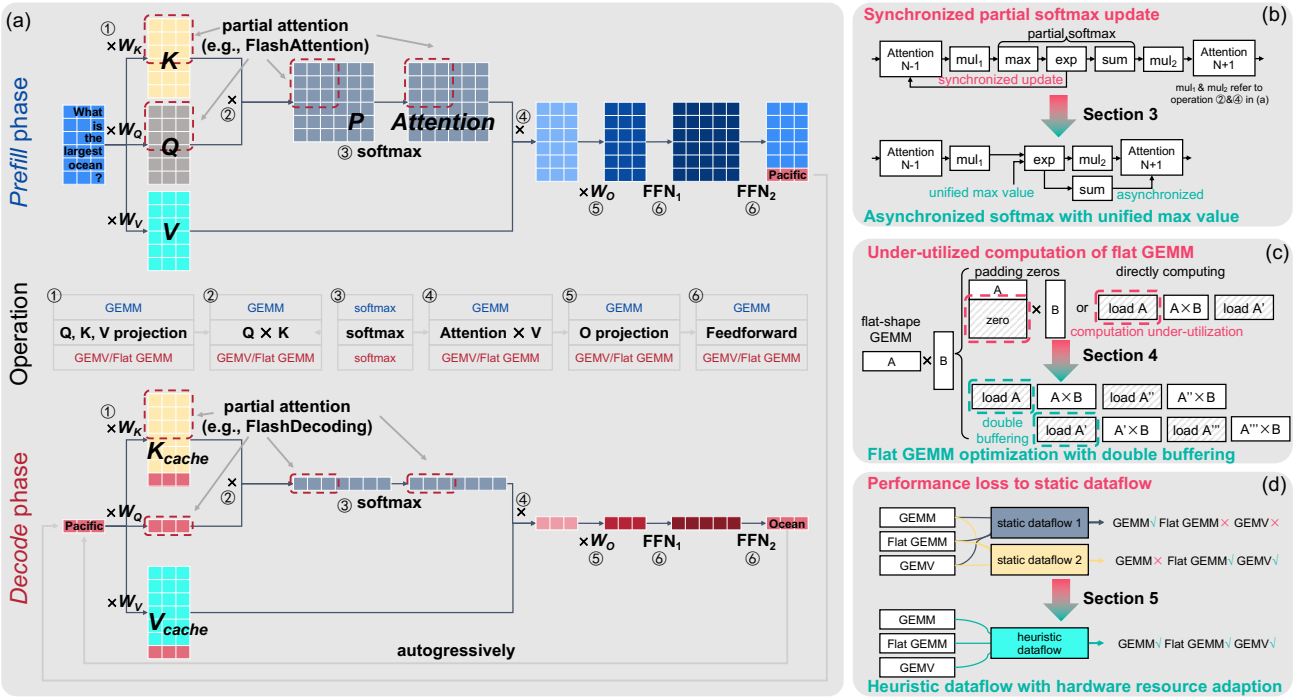


Figure 2. Overview of Large Language Model inference dataflow. *FlashDecoding++* proposes three solutions for corresponding challenges in Large Language Model inference. (a) The dataflow comparison between the *prefill* phase and the *decode* phase. The *prefill* phase mainly involves the GEMM operation, while the *decode* phase mainly involves the GEMV/Flat GEMM operation. (b) *FlashDecoding++* proposes the asynchronized softmax with unified max value technique, avoiding synchronized update to previous partial attention results. (c) *FlashDecoding++* optimizes flat GEMM by improving computation utilization. (d) *FlashDecoding++* heuristically optimizes dataflow.

million per day for the necessary computing hardware (DYLAN PATEL, 2023). Thus, optimizations on LLM inference performance will have a huge impact considering massive LLM inference scenarios. Many recent works have proposed techniques to accelerate LLM inference tasks, including DeepSpeed (Aminabadi et al., 2022), FlexGen (Sheng et al., 2023), vLLM (Kwon et al., 2023), OpenPPL (Sensetime, 2023a), FlashDecoding (Dao et al., 2023), TensorRT-LLM (Vaidya et al., 2023), and etc (Sensetime, 2023b; TGI, 2023; mlc, 2023; Sensetime, 2023a).

The LLM inference task generates tokens (*e.g.,* words) from the input sequence autoregressively, and can be organized into two typical phases: the *prefill* phase and the *decode* phase. The *prefill* phase generates the first token by processing the input prompt, and previous research (*e.g.,* FlashAttention (Dao et al., 2022; Dao, 2023)) optimizes latency for this phase. The *decode* phase generates the following tokens sequentially, and many works (Aminabadi et al., 2022; Sheng et al., 2023; Kwon et al., 2023; Sensetime, 2023b; Dao et al., 2023; Vaidya et al., 2023; Pham et al., 2023) focus on improving the throughput of generating tokens (*i.e.,* reducing latency of each token). The *prefill* phase dominates total time for scenarios of long-sequence input or generating short outputs (Dai et al., 2019; Dong et al.), while the *decode* phase constitutes a significant portion of the time when processing long output sequences (Xiao et al., 2023).

Figure 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 linear GEMM (General Matrix Multiplication) operations (*e.g., K, Q, V, O* weight projection and the feedforward) and the attention/softmax computation. For the attention computation, a softmax operation is adopted for a row in the attention matrix. To improve the parallelism, previous designs (Dao et al., 2022; 2023) divide the attention matrices into smaller tiles and rows are also split to compute partial softmax results. A synchronized softmax operation is adopted to update previous partial softmax results when a new partial softmax result is calculated. Such a **synchronized partial softmax update** accounts for 18.8% for the attention computation of Llama2-7B inference according to our profiling on NVIDIA Tesla A100 GPU with 1024 input length, resulting in the first challenge for accelerating LLM inference. Secondly, **the computation resources is under-utilized for the flat GEMM operation** during the *decode* phase. Because the *decode* phase sequentially generates tokens, the linear GEMM operation tends to be flat-shape (even turning into the GEMV (General Matrix-Vector Multiplication) operation when the batch size is 1). For the small batch size (*e.g.,* 8), previous designs (NVIDIA, 2017c;a) pad the matrix with zeros to perform GEMMs of larger sizes (*e.g.,* 64), leading to over 50% computation

under-utilization. Thirdly, **the performance of LLM inference suffers from the static dataflow** considering input dynamics and hardware configuration. For example, the small batch size makes the *decode* phase of LLM inference memory-bounded and the large batch size makes it compute-bounded. A single and static dataflow may lead to 50.25% performance loss for GEMMs of different shapes in LLM inference.

To tackle these challenges and enable a faster Large Language Model (LLM) inference, we present *FlashDecoding++* in this paper. *FlashDecoding++* creatively proposes the following contributions:

- **Asynchronized softmax with unified max value.** *FlashDecoding++* leverages a unified max value for different partial softmax computations. Each partial softmax result can be processed individually without synchronized update. Such a technique leads to 1.18× and 1.14× speedup for attention computation in the *prefill* stage and *decoding* stage, respectively.
- **Flat GEMM optimization with double buffering.** *FlashDecoding++* only pads the matrix size to 8 rather than 64 in previous designs for flat-shaped GEMM to improve computation utilization. We point out that flat GEMMs with different shapes face varied bottlenecks, and further improve the kernel performance by up to 52% with techniques like double buffering.
- **Heuristic dataflow with hardware resource adaption.** *FlashDecoding++* takes both input dynamics and hardware configurations into consideration and dynamically applies kernel optimization for the LLM inference dataflow. Such a technique leads to up to 29% speedup.

Because of the versatility of optimizations, the effectiveness of *FlashDecoding++* can be proved on both NVIDIA and AMD GPUs. *FlashDecoding++* achieves up to **4.86**× and **4.35**× speedup on both NVIDIA and AMD GPUs compared with Hugging Face implementations, respectively. Our extensive results show that *FlashDecoding++* achieves an average of **1.37**× speedup compared with FlashDecoding (Dao et al., 2023), a state-of-the-art LLM inference engine on various LLMs (*e.g.,* Llama2, ChatGLM2, etc.).

The rest of this paper is organized as follows. Section 2 introduces preliminaries of LLMs and related works on LLM inference acceleration. Our three techniques, the asynchronized softmax with unified max value, the flat GEMM optimization with double buffering, and the heuristic dataflow with hardware resource adaption are detailed in Section 3, 4, and 5, respectively. Section 6 presents the evaluation results. Related works on LLM inference are introduced in Section 7, and Section 8 concludes the paper.

2 BACKGROUND

2.1 LLM Inference Dataflow Overview

The task of LLM inference is to generate tokens from the input sequence, which can be used to complete a sentence or answer a question. An overview of the LLM inference dataflow is shown in Figure 2(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 Figure 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 autogressively, and each output token is used as the input token for the next The *decode* (*e.g.,* "Ocean" is further used as the input).

2.2 Operations in LLM Inference

The main operations in LLM inference are depicted as operation ① to ⑥ in Figure 2(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 (Vaswani et al., 2017), non-linear activation (Nair & Hinton, 2010; Ramachandran et al., 2017), mask (Vaswani et al., 2017), and others are not shown in the figure. Operations in the *prefill* phase and the *decode* phase are different in the shape of data. Because only one token (batch size=1) or few tokens (batch size>1) are processed at one time, **input matrices in the *decode* phase are flat-shape matrices or even vectors.**

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

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

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 Figure 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

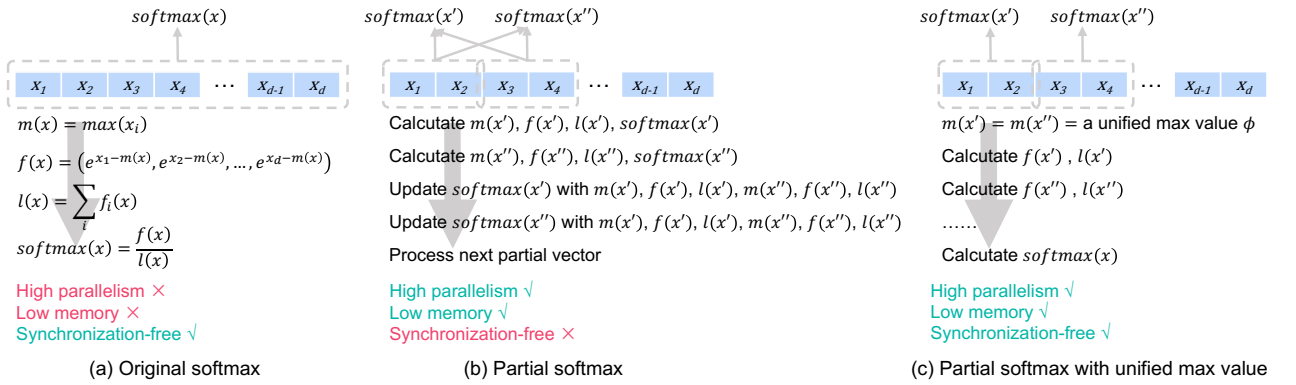


Figure 3. Comparison of different softmax computation schemes. (a) Softmax computation for the whole vector. (b) Computing partial softmax for each partial vector, and a synchronized update operation is required for all partial softmax results. (c) Computing partial softmax using a unified max value,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

exponents (*i.e.,* $l(x)$) to get the softmax result.

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

2.3 Attention Optimization

The softmax operation shown in Figure 3(a) requires all global data to be calculated and stored before it can proceed. This results in high memory consumption and low parallelism. Latter works propose the partial softmax technique to reduce memory consumption (Dao et al., 2022; Dao, 2023) or improve parallelism (Dao et al., 2023). Figure 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 Figure 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 for Llama2-7B inference on NVIDIA Tesla A100 GPU with 1024 input length.

3 Asynchronized Softmax with Unified Maximum Value

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

Challenge. The reason that synchronization is required lies in that the maximum value of each partial vector is different. The maximum value is used to avoid overflow of the exponent operation ($f(x)$ in Figure 3(a)), and exponents are summed ($l(x)$ in Figure 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 Figure 3(a)). Our key insight is, **the scaling factor can be an arbitrary number** rather than using the maximum value mathematically, shown in Equation (3). When we set $\phi = 0$, it becomes the original softmax computation (Bridle, 1989).

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

However, the scaling factor cannot be an arbitrary number considering the overflowing of the exponent computation.

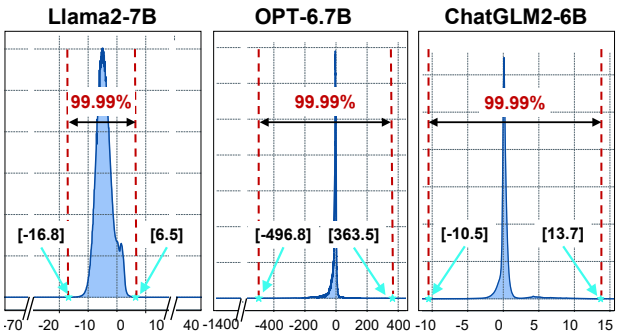


Figure 4. The statistical distribution of x_i (elements in the input vectors of softmax) in typical LLMs with different inputs.

For the case where $x_i \gg \phi$, $e^{x_i-\phi}$ overflows and cannot be represented using a fix-width floating point number (*e.g.,* `float32` for exponent results in current LLM engines). For another case where $x_i \ll \phi$, $e^{x_i-\phi} \to 0$, leading to precision loss. Thus, a proper scaling factor ϕ should be carefully selected to avoid the two cases above. Figure 4 shows the statistical distribution of x_i (elements in the input vectors of softmax) in typical LLMs with different inputs (Merity et al., 2016). Our key insight is, $> 99.99\%$ x_i **are within a certain range**. Specifically, for Llama2-7B, we have $-16.8 < x_i < 6.5$ for $> 99.99\%$ x_i. Because e^{b-a} and e^{a-b} can be represented by a `float32` format, we can set $\phi = a$ in Equation (3). For OPT-6.7B, we do not apply the technique in this section because of the large range in Figure 4.

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:

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

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

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 $x_i - \phi \le a$ or

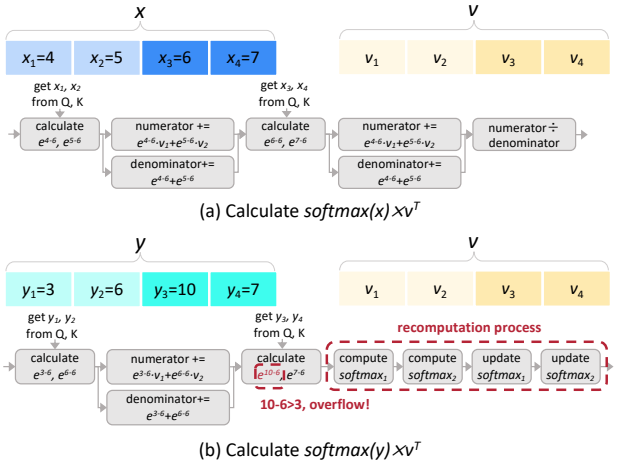


Figure 5. Example of asynchronized partial softmax computation. (a) Each partial softmax result is process individually without the synchronized update. (b) The recomputation process for all parital softmax computation is required when overflow happens.

$x_i - \phi \ge b$, the asynchronized 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 (Dao et al., 2022; Dao, 2023) and FlashDecoding (Dao et al., 2023)) shown in Figure 3(b). Such a recomputation scheme avoids overflow while introducing negligible overheads based on the statistical data shown in Figure 4.

Example. Figure 5 shows an example of the asynchronized 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 asynchronized threads. Then, each thread moves to the next partial vector for the corresponding computation (*i.e.,* $e^{x_3-\phi} \cdot v_3 + e^{x_4-\phi} \cdot v_4$ and $e^{x_3-\phi} + e^{x_4-\phi}$). Two threads are synchronized when all partial vectors are processed, and perform the division operation in Equation (4). For y, the first partial vector is processed similarly. However, we find that $y_3 - \phi > b$, then two threads are terminated and the first thread recomputes all partial vectors according to the synchronized partial softmax scheme in Figure 3(b).

4 Flat GEMM Optimization with Double Buffering

Motivation. The process of the *decode* phase is mainly composed of GEMV (batch size=1) or flat GEMM (batch size>1) operation. Without loss of generality, GEMV/GEMM operations can be represented using M, N, K, where the sizes of two multiplied matrices are $M \times K$ and $K \times N$. Tiling is a common technique for computing GEMMs on GPUs. The original matrices are

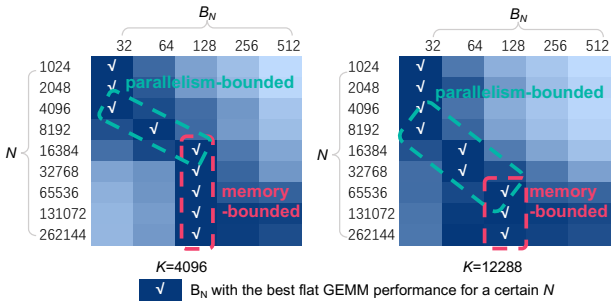


Figure 6. Normalized flat GEMM performance under different $N-$dimension sizes and $N-$dimension tiling sizes. We set $M = 8$ and execute GEMM on the NVIDIA Tesla A100 GPU.

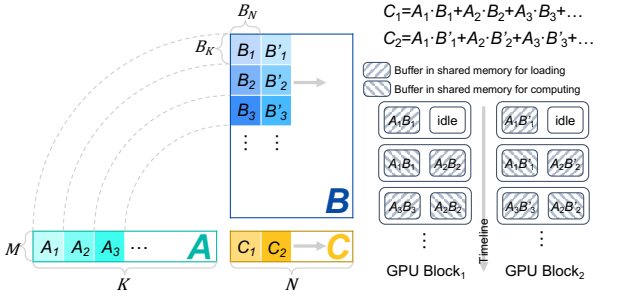


Figure 7. Double buffering for flat GEMM when $N-$dimension is large. The $M-$ dimension is padded to 8 and not tiled.

tiled into multiple sub-matrices, and then distributed across different computing units to enable parallel processing. Previous LLM inference engines utilize Tensor Core to accelerate these operations using libraries like cuBLAS (NVIDIA, 2017c) and CUTLASS (NVIDIA, 2017a). Although modern Tensor Core architectures (NVIDIA, 2023) process GEMM with $M = 8$, these libraries usually tile the $M-$dimension to 64 to hide memory latency. However, for GEMV or flat GEMM operations in the *decode* phase, we usually have $M \ll 64$ and the $M-$dimension is padded to 64 with zeros. The padding leads to under-utilized computation, and **the key problem is to process GEMV or flat GEMM operations with smaller tiles (*i.e.*, padding to 8 corresponding to modern Tensor Core architectures) in the $M-$dimension**.

Challenge. Processing GEMV or flat GEMM operations is non-trivial when the $M-$dimension is padded to 8. The tiling technique in modern libraries like cuBLAS (NVIDIA, 2017c) and CUTLASS (NVIDIA, 2017a) 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:

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

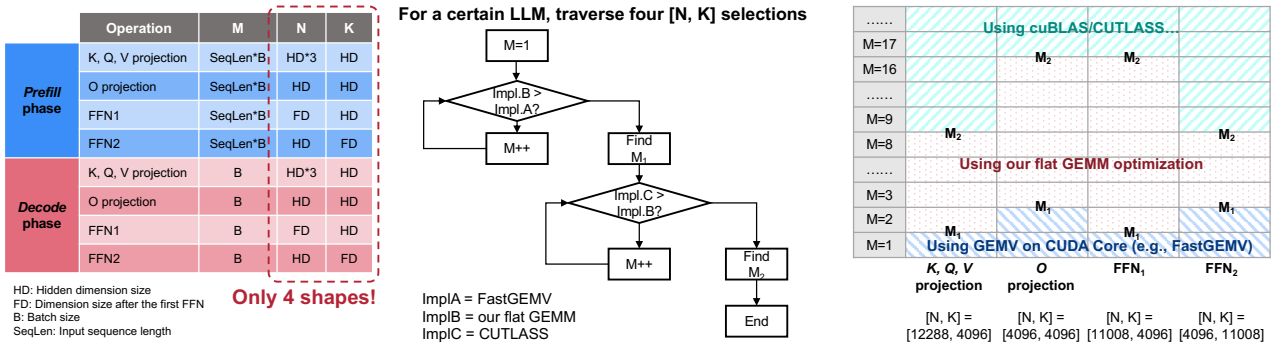
On the other hand, the parallelism is $\frac{N}{B_N}$. Thus, the computation/memory ratio shows a positive correlation with B_N while the parallelism shows a negative correlation with B_N, exposing a contradiction on improving the performance of GEMV or flat GEMM. We depict the normalized performance of the flat GEMM in Figure 6 with different N and B_N. Our key insight is, **for the smaller N, the flat GEMM is parallelism-bounded**. There are 108 Streaming Multiprocessors (SMs) in the NVIDIA Tesla A100. $\frac{N}{B_N}$ tends to be a constant (*e.g.,* 128 or 256), which is related to the hardware parallelism (number of SMs). Another key insight is, **for the larger N, the flat GEMM becomes memory-bounded**. The performance of these cases can be improved by hiding memory access latency.

Approach: Double Buffering. In order to hide memory access latency, we introduce the double buffering technique. for the flat GEMM operation. We allocate two separate buffers in the shared memory. The tile in one buffer performs the GEMM operation, while another buffer loads a new tile for the next GEMM operation. Thus, the computation and the memory access are overlapped. We apply such a technique when N is large in our practice.

Example. Figure 7 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.

5 Heuristic Dataflow with Hardware Resource Adaption

Motivation. Although *FlashDecoding++* optimizes the flat GEMM operation in Section 4, it does not cover all operations (even only for GEMMs) in the LLM inference. As mentioned in Figure 2(a), the shapes of GEMMs in dif-



(a) Different shapes of GEMMs in LLM (b) Decision flow (c) Example of heuristic dataflow with hardware resource adaption

Figure 8. Heuristic dataflow with hardware resource adaption in *FlashDecoding++*. (a) Only four $[N, K]$ shapes exist for a certain LLM. (b) The decision flow. We traverse all $[N, K]$ selections and profile the performance of three representative implementations. M is increased to find two inflection points for runtime heuristic dataflow. (c) *FlashDecoding++* heuristically utilizes Tensor Core/CUDA Core with the corresponding GEMV/GEMM implementation by referring to a lookup table.

ferent operations and two phases vary. Thus, the GEMM workload in the LLM inference can be GEMV (batch size=1 for the *decode* phase), flat GEMM (small batch size for the *decode* phase and short sequence length for the *prefill* phase) and conventional GEMM (large batch size or long sequence length for the *prefill* phase). In order to leverage the powerful computational ability of Tensor Core, current frameworks like FasterTransformer (NVIDIA, 2017b) and DeepSpeed (Aminabadi et al., 2022) tend to utilize the highly optimized GEMM implementation from cuBLAS (NVIDIA, 2017c) to deal with different workloads. However, the Tensor Core implementation fails with the GEMV workload. The GEMV workload can be optimized by utilizing CUDA Core in previous designs like FastGEMV (Wang, 2023). For a Llama2-7B linear layer in the *decode* phase, the Tensor Core implementation from cuBLAS only achieves 82.15% of the performance of CUDA Core implementation using FastGEMV on an NVIDIA A100 GPU. On the other hand, using CUDA Core to do the projection on a batch-size=4 decoding input only achieves 49.75% performance compared with the Tensor Core implementation. Thus, in order to approach the optimal computation performance, **a heuristic dataflow is supposed to be exploited for different workloads.**

Challenge. Although a heuristic dataflow potentially exists in the implementation of different linear workloads, it is challenging to build the mapping from a certain workload to an optimal implementation. In the scenario of LLM inference, there are various factors that influence the implementation performance of linear workloads: (a) Input dynamics. The variety of the batch size and the input sequence length brings dynamic workloads. (b) Model diversity. The linear workload varies with different model structures and sizes. (c) GPU capacities. The relative performance between implementations changes with GPU characteristics, such as memory bandwidth, cache size, and computational ability. (d) Engineering effects. The engineering effort also highly impacts the kernel performance. All these influential factors build a large search space, making it non-trivial to generate an effective mapping between the linear workload and the corresponding optimal implementation.

Analysis and Insights. Although all influential factors form a large search space, the homogeneity of different layers in LLM significantly reduces the search space for operator optimization. Figure 2(a) shows four linear GEMV/GEMM operations in the *prefill* phase and the *decode* phase, *i.e.*, K, Q, V projection, O projection, and two feedforward operations. Each GEMV/GEMM operation can be can be abstracted as a multiplication between an $(M \times K)$-shaped matrix and a $(K \times N)$-shaped matrix. Our key insight is, **there are only four $[K, N]$ shapes for a certain LLM.** Moreover, M is only related to the input sequence length and the batch size for the *prefill* phase, and the batch size for the *decode* phase. Figure 8(a) shows limited shapes of GEMV/GEMM operations in the LLM inference.

Approach: Decision flow for inflection points. Because only four $[K, N]$ shapes exist for a certain LLM, we use three types of implementations for GEMV/GEMM operations when M varies: FastGEMV for the GEMV and flat GEMM operations (ImplA), our flat GEMM optimization in Section 4 (ImplB), and the CUTLASS (NVIDIA, 2017a) libraries optimized for the conventional GEMM (ImplC). Thus, it is important to decide whether applying ImplA or ImplB for a small M, and ImplB or ImplC for a large M. Figure 8(b) shows the decision flow. *FlashDecoding++* profiles the performance of ImplA and ImplB for a certain M, and increases M to find an inflection point M_1 where the performance of ImplB is better than ImplA. Another inflection point M_2 is found similarly where the performance of ImplC is better than ImplB. Note that each $[N, K]$ gets its individual M_1 and M_2.

Approach: Heuristic dataflow. For the runtime LLM inference, *FlashDecoding++* adopts ImplA using CUDA

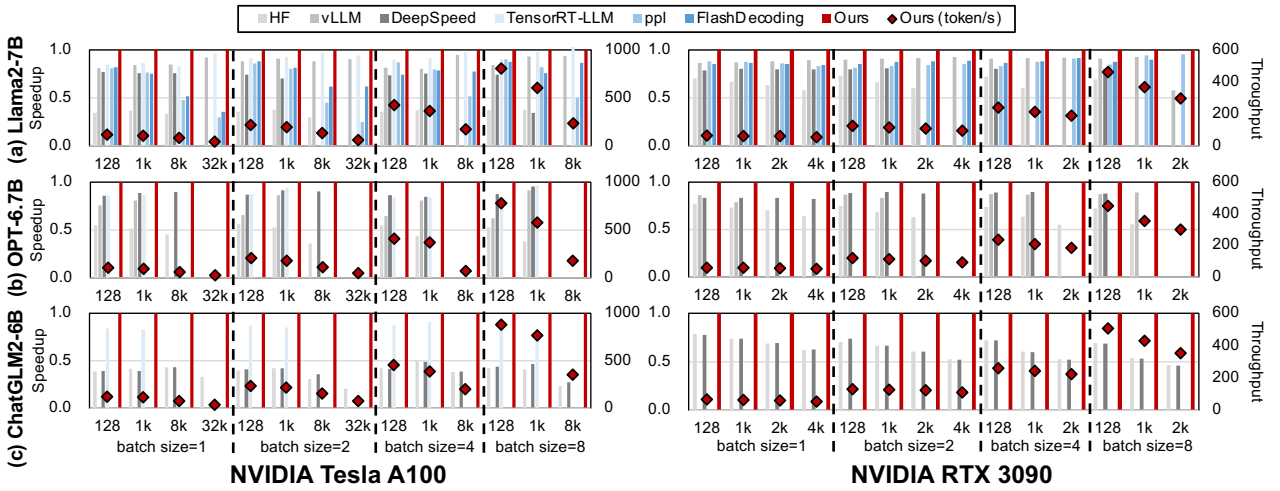


Figure 9. Speedup of the *decode* phase on NVIDIA GPUs, normalized to *FlashDecoding++*. Blank bars represent the model cannot be executed: (1) Hugging Face and DeepSpeed run out of memory with long sequences. (2) vLLM does not support OPT-6.7B with sequence length $>$ 2k and ChatGLM2-6B. (3) TensorRT-LLM fails to compile for OPT-6.7B and ChatGLM2-6B with sequence length $>=$ 8k. (4) FlashDecoding and ppl only supports Llama2 models.

Table 1. Hardware Platforms

	NVIDIA		AMD	
GPU	Tesla A100 80 GB CUDA 12.1	RTX3090 24 GB CUDA 11.6	MI210 64GB ROCm 5.7	RX7900XTX 24GB ROCm 5.6
CPU	Intel Xeon Silver 8358P 2.60 GHz	Intel Xeon Gold 6226R 2.90GHz	AMD EPYC 7K62 2.60GHz	Intel Core i9-10940X 3.30GHz

Table 2. Model Configuration

Model	Dimension	Heads	Layers	Context Length
Llama2-7B	4096	32	32	4k
Llama2-13B	5120	40	40	4k
OPT-6.7B	4096	32	32	2k
ChatGLM2-6B	4096	32	32	32k

Core when $M < M_1$, and ImplB/ImplC using Tensor Core when $M_1 \leq M < M_2/M_2 \leq M$. Note that the decision flow are executed offline, it does not affect the performance of runtime LLM inference.

Example. Figure 8(c) shows an example of applying the heuristic dataflow for the Llama2-7B model. Four $[N, K]$ shapes are [12288, 4096] for K, Q, V projection, [4096, 4096] for O projection, [11008, 4096] and [4096, 11008] for FFN. For each $[N, K]$, the inflection points are found based on the decision flow in Figure 8(c). Then, a lookup table is formed, and each GEMV/GEMM operation is executed according to corresponding implementations during runtime. In this example, FastGEMV is adopted for the K, Q, V projection when batch size=1 ($M = 1$) for the *decode* phase, and our flat GEMM optimization is applied when batch size=1/input sequence length=8 for FFN_1 ($M = 8$).

6 Evaluation

6.1 Experiments Setup

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

6.1.1 Hardware Platforms

We evaluate the performance of *FlashDecoding++* and other LLM engines on both NVIDIA and AMD platforms to make a comprehensive comparison. We choose two different GPUs for each platform: Tesla A100 and RTX3090 for NVIDIA, MI210 and RX7900XTX for AMD. We show the detailed configuration in Table 1.

6.1.2 LLM Engine Baselines

We implement our *FlashDecoding++* using the Pytorch-based front-end with the C++ and CUDA backend for NVIDIA GPUs while ROCm for AMD GPUs. We compare the inference performance in both *prefill* phase and *decode* phase with the following LLM engine baselines: Hugging Face (HF) v4.34.1 (Wolf et al., 2020), vLLM v0.1.7 (Kwon et al., 2023), DeepSpeed v0.11.1 (Aminabadi et al., 2022), TensorRT-LLM v0.5.0 (Vaidya et al., 2023), OpenPPL (Sensetime, 2023a), FlashAttention v2.3.5 (Dao, 2023) and FlashDecoding (Dao et al., 2023). These baselines are introduced in Section 7.

6.1.3 Models

We evaluate the performance of *FlashDecoding++* with other LLM inference engines on three typical Large Language Models: Llama2, OPT, and ChatGLM2. Table 2

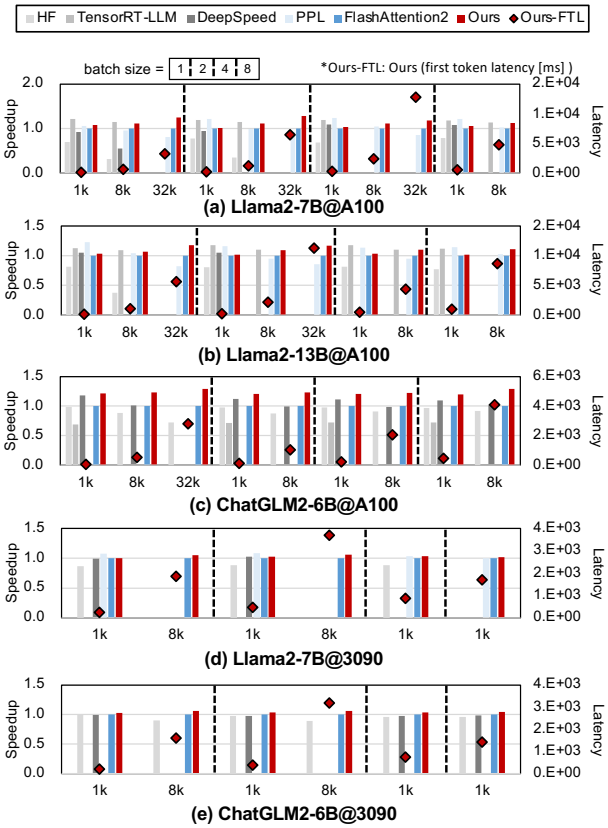


Figure 10. Speedup of the *prefill* phase on NVIDIA GPUs, normalized to FlashAttention. Blank bars represent failed execution: (1) Hugging Face, DeepSpeed and TensorRT-LLM run out of memory with long sequences. (2) vLLM does not support ChatGLM2-6B. (3) TensorRT-LLM fails to compile on RTX 3090 GPUs with 24GB memory, and fails to compile for ChatGLM2-6B with sequence length >= 8k. (4) ppl only supports Llama2 models.

shows the detailed configuration of these models. Note that there may be several models in one LLM (*e.g.,* Llama2-7B, Llama2-13B) with different configurations (*e.g.,* number of heads and layers).

- **Llama2** (Touvron et al., 2023) is a mainstream open-source LLM set released by Meta in 2023. It is a collection of pretrained and fine-tuned generative text models ranging in scale from 7B to 70B parameters.
- **OPT** (Zhang et al., 2022), is a suite of decoder-only pre-trained transformers ranging from 125M to 175B parameters released by Meta AI.
- **ChatGLM2** (Du et al., 2022) is an open-source LLM supporting bilingual (Chinese-English) chat.

6.2 Comparison with State-of-the-art

We compare *FlashDecoding++* with state-of-the-art LLM inference engines in Figure 9 and Figure 10 on NVIDIA GPUs, Figure 11 and Figure 12 for AMD GPUs. For the

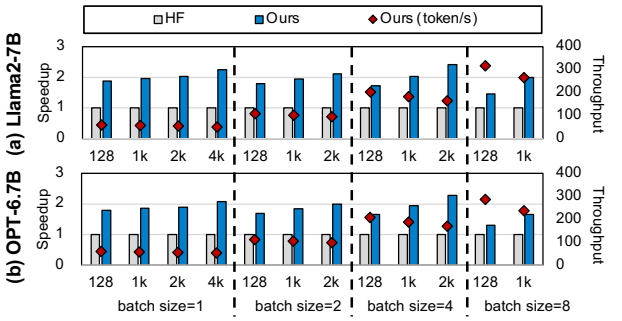


Figure 11. Speedup of the *decode* phase on AMD RX7900XTX.

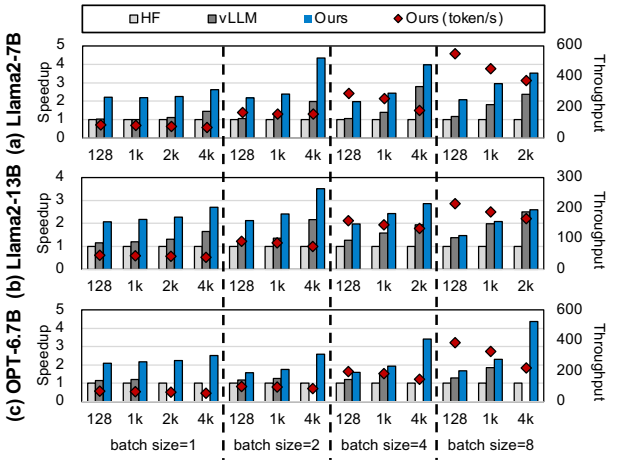


Figure 12. Speedup of the *decode* phase on AMD MI210. There are blank bars for vLLM because it doesn't support sequence length over 2k for OPT-6.7B.

decode phase, *FlashDecoding++* achieves up to **4.86**× speedup compared with Hugging Face implementations on three LLMs and two GPUs. The average speedup over vLLM, DeepSpeed, TensorRT-LLM, OpenPPL, and FlashDecoding is 1.24×, 1.44×, 1.13×, 1.24×, and 1.21× (**1.37**× on Tesla A100 compared with FlashDecoding), respectively. For the *prefill* phase, *FlashDecoding++* achieves up to 1.40× speedup compared with Hugging Face implementations. The average speedup over DeepSpeed, TensorRT-LLM, OpenPPL, FlashAttention2 and FlashDecoding is 1.05×, 1.06×, 1.08×, 1.09×, and 1.08×, respectively. For *prefill* phase, *FlashDecoding++* performs worse than some baselines with short sequences but always gains speedup with long sequences. The reason is that, for *prefill* phase, we only optimize the attention operation, and the attention operation occupies more of the latency as sequence length grows.

We also show the *decode* results on two AMD GPUs. Currently, only Hugging Face and vLLM can be executed on AMD GPUs as the baselines, and vLLM does not support RX7900XTX yet. *FlashDecoding++* achieves up to 2.41× and **4.35**× compared with Hugging Face on RX7900XTX and MI210, respectively. And on MI210, the average speed of *FlashDecoding++* compared to vLLM is 1.86×.

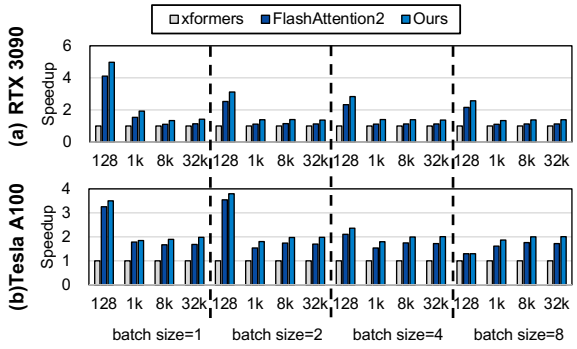


Figure 13. Benefits of asynchronized softmax (*prefill* phase).

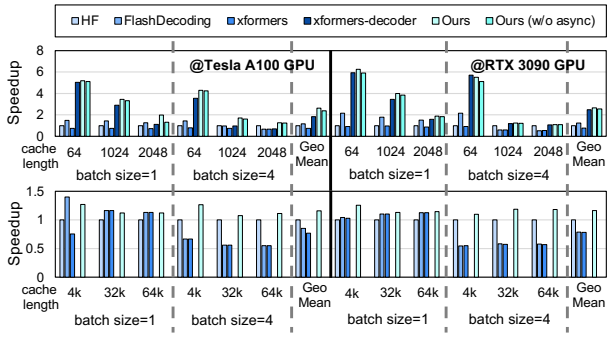


Figure 14. Benefits of asynchronized softmax (*decode* phase).

6.3 Ablation Studies

6.3.1 *Asynchronized Softmax Computation*

Benefits. The asynchronized softmax scheme can be applied to both the *prefill* phase and the *decode* phase. We test the proposed scheme against state-of-the-art attention implementations in Figure 13 and Figure 14 on NVIDIA GPUs. For the *prefill* phase, *FlashDecoding++* achieves 1.52× and 1.19× average speedup compared with xformers (Lefaudeux et al., 2022) and FlashAttetion2. For the *decode* phase, *FlashDecoding++* outperforms the decoding-tailored implementation of xformers (denoted as xformers-decoder in Figure 14) with short KV cache length, and achieves up to 2.02× speedup over FlashDecoding with long context.

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. 3, we introduce a recomputation mechanism into the asynchronized softmax, which automatically selects FlashAttention for computation when the intermediate results overflow. The frequency of recomputation is statistically obtained to be 0.45% on average across datasets including ARC (Clark et al., 2018), HellaSwag (Zellers et al., 2019) and Winogrande (Sakaguchi et al., 2019).

Scalability. We extend our approach to models including CodeLlama-7B (Rozière et al., 2023) and Vicuna-7B (Chiang et al., 2023), which are fine-tuned on Llama2-7B 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 asynchronized softmax method is also applicable to those fine-tuned models.

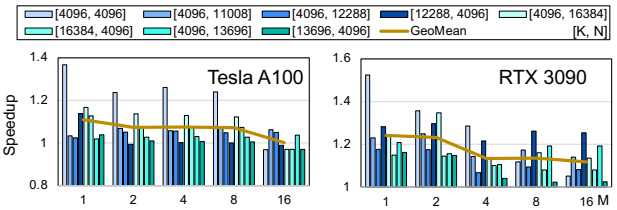


Figure 15. Speedup over cuBLAS with flat GEMM optimization.

6.3.2 *Flat GEMM Optimization*

Benefits. We test our flat GEMM kernel performance with 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 three LLMs (Llama2-7B, OPT-6.7B, and ChatGLM2-6B) are depicted in Figure 15. The flat GEMM optimization in *FlashDecoding++* 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 purpose, 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.

Scalability. 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 15 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. In fact, on AMD GPUs, we significantly rely on heuristics to achieve performance gains.

6.3.3 *Benefits of Heuristic Dataflow*

We test speedup of the *decode* phase by adopting the heuristic dataflow in three LLMs (Llama2-7B, OPT-6.7B, and ChatGLM2-6B) on NVIDIA GPUs, and two LLMs (Llama2-7B, OPT-6.7B) on AMD GPUs. The input length is set to 1024, and the results are shown in Figure 16. The heuristic dataflow achieves an average of 10% and 20% (up to 29%) speedup on Tesla A100 and RTX 3090, respectively. On AMD GPUs, the extension of FastGEMV implementa-

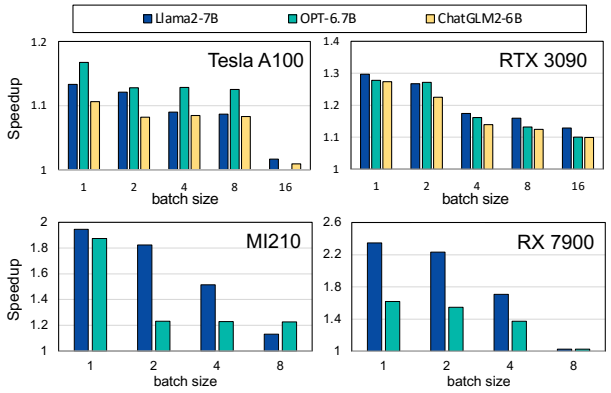


Figure 16. Benefits of the heuristic dataflow (input length=1024).

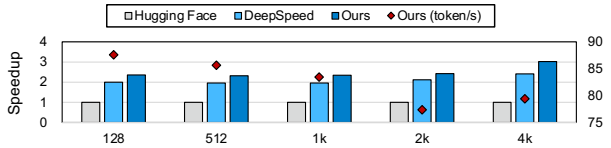


Figure 17. Performance on Llama2-13B on two Tesla A100 GPUs.

tion proves to be highly efficient, and leads to significant performance gains with small batch sizes. The average speedup of using heuristics is 57% and 37% on MI210 and RX7900XTX, respectively.

6.4 Multi-GPUs Performance

FlashDecoding++ supports executing large LLMs on multiple GPUs. We use Llama2-13B running on 2 NVIDIA Tesla A100 GPUs to evaluate the performance of *FlashDecoding++*. The result in Figure 17 shows that, *FlashDecoding++* achieves 2.48× and 1.19× higher *decode* phase throughput compared with Hugging Face (Wolf et al., 2020) and DeepSpeed (Aminabadi et al., 2022).

7 Related Works

Large language model inference acceleration has gained significant attention in recent research, with several notable approaches and techniques emerging in the field. **DeepSpeed** (Aminabadi et al., 2022) 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 KV-cache. **vLLM** (Kwon et al., 2023) 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** (Dao et al., 2022; Dao, 2023) optimizes the self-attention computation process during the prefill phase through improved parallelism and workload distribution. **FlashDecoding** (Dao et al., 2023) is an extension of FlashAttention and enhances the parallelism through spliting K and V, supporting efficient self-attention computation for long sequence during the decode phase. **FasterTransformer** (NVIDIA, 2017b) and **OpenPPL** (Sensetime, 2023a) implement large model inference engines using *C++* to reduce overhead resulting from kernels scheduling, compared to *Python* implementations. They also employ memory management techniques and kernel fusion to achieve efficient LLM inference. **TensorRT-LLM** (Vaidya et al., 2023) is built upon the *TensorRT* (NVIDIA) and the *FasterTransformer* (NVIDIA, 2017b) engine (*C++*) and incorporates cutting-edge open-source technologies such as *FlashAttention* (Dao et al., 2022; Dao, 2023). Additionally, it enhances its ease of use by providing the *Python API*.

8 Conclusion

We propose *FlashDecoding++*, a fast Large Language Model inference engine in this paper. *FlashDecoding++* accelerates mainstream LLMs with multiple hardware backend support. *FlashDecoding++* proposes three novel designs: the asynchronized softmax with unified max value, the flat GEMM optimization with double buffering, and the heuristic dataflow with hardware resource adaption, achieving up to **4.86**× and **4.35**× speedup on NVIDIA and AMD GPUs compared with Hugging Face implementations. *FlashDecoding++* also achieves an average of **1.37**× speedup compared with state-of-the-art LLM inference engines, FlashDecoding, on various LLMs.

References

Text generation inference: Fast inference optimize for llms. [Online], 2023. https://github.com/huggingface/text-generation-inference/.

Mlc llm: Machine learning compilation for large language models. [Online], 2023. https://github.com/mlc-ai/mlc-llm.

AMD. Tools to translate cuda source code into portable hip c++ automatically. [Online], 2023. https://github.com/ROCm-Developer-Tools/HIPIFY.

Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. IEEE, 2022.

Anil, R., Dai, A. M., Firat, O., Johnson, M., Lepikhin, D., Passos, A., Shakeri, S., Taropa, E., Bailey, P., Chen, Z.,

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

Bridle, J. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. *Advances in neural information processing systems*, 2, 1989.

Chiang, W.-L., Li, Z., Lin, Z., Sheng, Y., Wu, Z., Zhang, H., Zheng, L., Zhuang, S., Zhuang, Y., Gonzalez, J. E., Stoica, I., and Xing, E. P. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL https://lmsys.org/blog/2023-03-30-vicuna/.

Clark, P., Cowhey, I., Etzioni, O., Khot, T., Sabharwal, A., Schoenick, C., and Tafjord, O. Think you have solved question answering? try arc, the ai2 reasoning challenge. *ArXiv*, abs/1803.05457, 2018.

Clusmann, J., Kolbinger, F. R., Muti, H. S., Carrero, Z. I., Eckardt, J.-N., Laleh, N. G., Löffler, C. M. L., Schwarzkopf, S.-C., Unger, M., Veldhuizen, G. P., et al. The future landscape of large language models in medicine. *Communications Medicine*, 3(1):141, 2023.

Cui, C., Ma, Y., Cao, X., Ye, W., and Wang, Z. Receive, reason, and react: Drive as you say with large language models in autonomous vehicles. *arXiv preprint arXiv:2310.08034*, 2023.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., and Salakhutdinov, R. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

Dao, T., Haziza, D., Massa, F., and Sizov, G. Flash-decoding for long-context inference. [Online], 2023. https://crfm.stanford.edu/2023/10/12/flashdecoding.html.

Dong, Z., Tang, T., Li, L., and Zhao, W. A survey on long text modeling with transformers. arxiv 2023. *arXiv preprint arXiv:2302.14502*.

Du, Z., Qian, Y., Liu, X., Ding, M., Qiu, J., Yang, Z., and Tang, J. Glm: General language model pretraining with autoregressive blank infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 320–335, 2022.

DYLAN PATEL, A. A. The inference cost of search disruption - large language model cost analysis. [Online], 2023. https://www.semianalysis.com/p/the-inference-cost-of-search-disruption.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Lefaudeux, B., Massa, F., Liskovich, D., Xiong, W., Caggiano, V., Naren, S., Xu, M., Hu, J., Tintore, M., Zhang, S., Labatut, P., and Haziza, D. xformers: A modular and hackable transformer modelling library. https://github.com/facebookresearch/xformers, 2022.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models, 2016.

Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

Nerdynav. Up-to-date chatgpt statistics and user numbers [oct 2023]. [Online], 2023. https://nerdynav.com/chatgpt-statistics.

NVIDIA. Nvidia tensorrt: An sdk for high-performance deep learning inference. [Online]. https://developer.nvidia.com/tensorrt.

NVIDIA. Cutlass: Cuda templates for linear algebra subroutines. [Online], 2017a. https://github.com/NVIDIA/cutlass.

NVIDIA. Fastertransformer: About transformer related optimization, including bert, gpt. [Online], 2017b. https://github.com/NVIDIA/FasterTransformer.

NVIDIA. cublas: Basic linear algebra on nvidia gpus. [Online], 2017c. https://developer.nvidia.com/cublas.

NVIDIA. Nvidia tensor core. [Online], 2023. https://www.nvidia.com/en-us/data-center/tensor-cores/.

OpenAI. Openai pricing. [Online], 2023. https://openai.com/pricing.

Pham, A., Yang, C., Sheng, S., Zhao, S., Lee, S., Jiang, B., Dong, F., Guan, X., and Ming, F. OpenLLM: Operating LLMs in production, June 2023. URL https://github.com/bentoml/OpenLLM.

Ramachandran, P., Zoph, B., and Le, Q. V. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Sakaguchi, K., Bras, R. L., Bhagavatula, C., and Choi, Y. Winogrande: An adversarial winograd schema challenge at scale. *arXiv preprint arXiv:1907.10641*, 2019.

Sensetime. Openppl: A high-performance deep learning inference platform. [Online], 2023a. https://openppl.ai/home.

Sensetime. A light and fast inference service for llm. [Online], 2023b. https://github.com/ModelTC/lightllm.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Re, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single gpu. 2023.

Thirunavukarasu, A. J., Ting, D. S. J., Elangovan, K., Gutierrez, L., Tan, T. F., and Ting, D. S. W. Large language models in medicine. *Nature medicine*, 29(8):1930–1940, 2023.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Vaidya, N., Oh, F., and Comly, N. Optimizing inference on large language models with nvidia tensorrt-llm, now publicly available. [Online], 2023. https://github.com/NVIDIA/TensorRT-LLM.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Wang, S. Fastgemv: High-speed gemv kernels. [Online], 2023. https://github.com/wangsiping97/FastGEMV.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://aclanthology.org/2020.emnlp-demos.6.

Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., and Choi, Y. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer, L. Opt: Open pre-trained transformer language models, 2022.

A DETAILED LLM DATAFLOW IN *FlashDecoding++* WITH KERNEL FUSION

FlashDecoding++ utilizes open-source kernels and fuses kernels in LLMs. Element-wise kernels like activation and

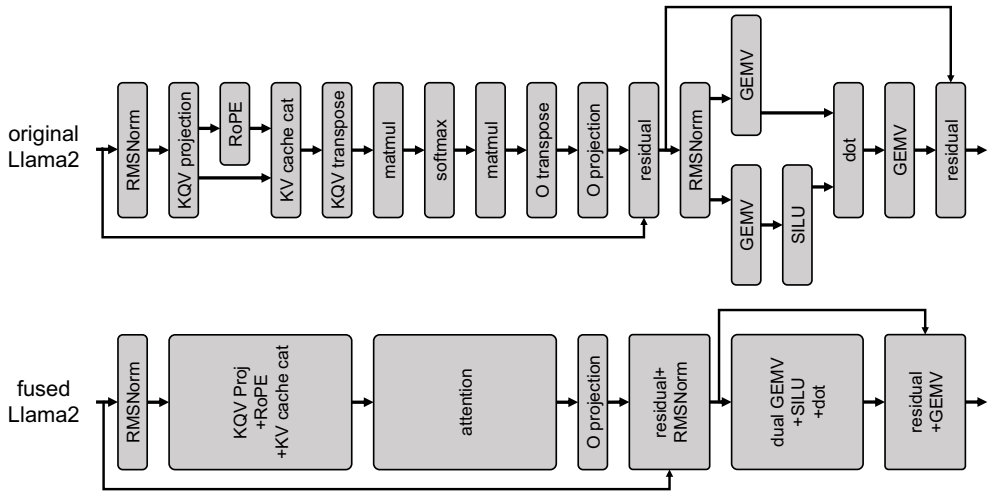


Figure 18. Example of kernel fusion of Llama2 dataflow.

position encoding are fused with linear kernels. We show an example of kernel fusion of Llama2 dataflow compared to the original dataflow in Figure 18.

B IMPLEMENTATION ON AMD

Due to the PyTorch's support for AMD GPUs, we can perform large language model inference on AMD GPUs similar to what we do on NVIDIA GPUs. We have implemented and validated the effectiveness of our proposed methods on AMD GPUs using AMD parallel programming. AMD parallel programming shares many similarities with NVIDIA parallel programming. Their programming models are divided into grid, block, warp, and thread. Similar to the CUDA runtime library of NVIDIA, AMD has the ROCm runtime library. We can use HIP to develop kernels for AMD GPUs. HIP has APIs that closely resemble CUDA APIs. We can easily port CUDA code developed for NVIDIA GPUs to HIP code for AMD GPUs by modifying the API names or using the HIPIFY tool (AMD, 2023). However, architectural differences between GPUs mean that efficient kernels developed for NVIDIA GPUs may not necessarily be efficient on AMD GPUs, and in some cases, they may not even run. For example, consumer-level GPU like the RX7900XTX, based on the RDNA3 architecture, lacks structures similar to the Tensor Core and cannot efficiently perform matrix operations using WMMA instructions as CUDA. In contrast, compute-level GPU like the MI210, based on the CDNA2 architecture, has the Matrix Core but with a warp size of 64, unlike NVIDIA GPUs. This necessitates optimizations tailored for each of these GPUs.

We employ different strategies for our implementations on these two types of AMD GPUs to accommodate their distinct characteristics compared to NVIDIA GPUs. Since our asynchronized softmax optimization for *decode* phase does not use the Tensor Core, we migrate CUDA codes to HIP and run them on these two types of AMD GPUs. However, the flat GEMM optimization uses the Tensor Core, so we need different implementation approaches for the RX7900XTX and MI210. Given that MI210 has Matrix Cores, a hardware structure similar to Tensor Cores for efficient matrix computation, we migrate CUDA code and adjust the warp size to 64 to suit this GPU. RX7900XTX does not have Matrix Cores, preventing direct code migration. To this end, we use the WMMA compiler intrinsics, such as `__builtin_amdgcn_wmma_f16_16x16x16_f16_w32`, to develop flat GEMM kernels resulting in 20% speedup than the torch.matmul used in PyTorch on the RX7900XTX.