PIM-SHERPA: Software Method for On-device LLM Inference by Resolving PIM Memory Attribute and Layout Inconsistencies

Sunjung Lee
Samsung Advanced Institute of Technology
Suwon, Republic of Korea
sj7748.lee@samsung.com

Sanghoon Cha
Samsung Advanced Institute of Technology
Suwon, Republic of Korea
s.h.cha@samsung.com

Hyeonsu Kim
Samsung Advanced Institute of Technology
Suwon, Republic of Korea
hyeonsu.kim@samsung.com

Seungwoo Seo
Samsung Advanced Institute of Technology
Suwon, Republic of Korea
sgwoo.seo@samsung.com

Yuhwan Ro
Samsung Advanced Institute of Technology
Suwon, Republic of Korea
yuhwan.ro@samsung.com

Sukhan Lee
Samsung Electronics
Hwaseong, Republic of Korea
sh1026.lee@samsung.com

Byeongho Kim
Samsung Electronics
Hwaseong, Republic of Korea
bh1122.kim@samsung.com

Yongjun Park
Yonsei University
Seoul, Republic of Korea
yongjunpark@yonsei.ac.kr

Kyomin Sohn
Samsung Electronics
Hwaseong, Republic of Korea
kyomin.sohn@samsung.com

Seungwon Lee
Samsung Advanced Institute of Technology
Suwon, Republic of Korea
seungw.lee@samsung.com

Jaehoon Yu
Samsung Advanced Institute of Technology
Suwon, Republic of Korea
jae-hoon.yu@samsung.com

Abstract

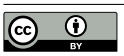
On-device deployment of large language models (LLMs) is rapidly expanding across mobile and edge platforms. LLM inference comprises a compute-intensive prefill phase and a memory bandwidth-intensive decode phase, with the latter widely recognized as well-suited for processing-in-memory (PIM) in both academia and industry. However, practical PIM-enabled systems face two primary obstacles between these phases. First, a memory attribute inconsistency arises because prefill favors placing weights in a cacheable region for reuse, whereas decode requires a non-cacheable region to trigger PIM execution. Second, a weight layout inconsistency exists between the host processor and PIM. To address these problems, we introduce *PIM-SHERPA*, a software-only method for efficient LLM inference on PIM-enabled systems. PIM-SHERPA introduces two approaches, DRAM double buffering (DDB) and online weight rearrangement with swizzled memory copy (OWR). DDB maintains a single copy of PIM-aware weights in the non-cacheable region while prefetching next-layer swizzled weights into small cacheable buffers. OWR performs on-demand swizzled memory copies immediately before GEMM operations. Evaluated on Llama 3.2, PIM-SHERPA achieves 47.8 - 49.7% memory capacity savings over a PIM emulation baseline while matching theoretical maximum performance. To the best of our knowledge, this is the first work to explicitly focus on memory attribute inconsistency and propose effective solutions on product-level PIM-enabled systems.

CCS Concepts

• **Hardware** → **Emerging architectures**; • **Software and its engineering** → **Software system structures**; • **Computer systems organization** → *Embedded software*; • **Computing methodologies** → *Neural networks*.

Keywords

Processing-in-memory (PIM), On-device LLM inference, Memory attribute inconsistency, Weight layout inconsistency, DRAM double buffering, Layout swizzling



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846830

1 Introduction

Large language models (LLMs) have rapidly advanced language understanding and generation, driving applications such as conversational assistants and real-time translation. As these services proliferate, there is strong demand to run inference on devices such as smartphones and edge systems to eliminate network latency and preserve user privacy. The major vendors have already offered LLM features on mobile platforms [2, 7, 30, 34], and lightweight runtimes are emerging to support such deployments [6, 38].

LLM inference consists of prefill and decode phases, which exhibit different operational characteristics. In a single-batch scenario [41], the prefill processes a significant number of tokens,

leading linear layers to be executed as matrix-matrix multiplication (GEMM). Because prefill is compute-intensive, it runs best on host processors such as GPUs, NPUs, or CPUs. The decode processes a single token at a time, which turns the linear layers into matrix-vector multiplication (GEMV) and shifts the bottleneck to the DRAM bandwidth. To alleviate the DRAM bandwidth bottleneck in on-device environments, processing-in-memory (PIM) architectures, specifically LPDDR-PIM, have recently emerged as a promising alternative [9, 13, 15, 18, 19] by utilizing up to eight times the effective DRAM bandwidth.

However, applying LPDDR-PIM to on-device inference introduces critical system-level challenges between the prefill and decode phases: 1) **memory attribute inconsistency** and 2) **weight layout inconsistency**. Memory attribute inconsistency arises because the weight requires contradictory memory attributes between phases. Prefill requires weights in a cacheable region to maximize cache reuse. Decode requires weights in a non-cacheable region to enable PIM. Specifically, if the weights are in a cacheable region, cache hits prevent memory requests from reaching the memory controller, effectively blocking the DRAM requests required to trigger PIM execution. The weight layout inconsistency is due to the different optimal memory layouts for each phase. The host-side operation prefers a contiguous host-friendly layout that benefits from channel and bank interleaving. In contrast, PIM prefers a PIM-aware layout that places data in contiguous DRAM columns within each DRAM bank to maximize the utilization of SIMD units coupled to the bank [13, 18]. These conflicting layout requirements arise when LLM inference performs both prefill and decode.

Previous studies on PIM-based acceleration of LLM inference [10, 13, 20, 28, 36] have not explicitly addressed the memory attribute and weight layout inconsistencies. HBM-PIM [23] simply duplicates weights by keeping a cacheable host-friendly copy and a non-cacheable PIM-aware copy, which nearly doubles memory footprint and narrows the set of models that can be run by on-device. FACIL [37] augments the memory controller to resolve the layout inconsistency so that both phases can access suitable layouts from a single copy of weights, but the approach requires hardware changes and leaves the memory attribute issue unresolved.

To address these challenges, we introduce a **s**oftware met**h**od for efficient on-device LLM inf**er**ence by resolving **P**IM memory **a**ttribute and layout inconsistencies, named **PIM-SHERPA**. Although this work focuses on LPDDR-PIM in mobile environments, the core software principles of PIM-SHERPA are architecture agnostic and broadly applicable to other PIM standards, such as HBM-PIM and GDDR-PIM, used in edge and cloud servers. We propose two software-only methods tailored to the trend toward longer input sequence lengths driven by interactive LLMs [14] and on-device retrieval-augmented generation (RAG) [29, 35]. DRAM double buffering (DDB) maintains two small DRAM buffers in the cacheable region. While the current layer executes GEMM using one cacheable buffer, the weight for the subsequent layer is concurrently prefetched into the alternate buffer. By fetching the weight in parallel with computation, the overheads associated with rearrangement and memory copy effectively overlap with the compute latency. A swizzled memory copy transforms weights from the PIM-aware layout to the host-friendly layout and stores them in the cacheable region. Online weight rearrangement with swizzled memory copy (OWR) assumes sufficiently long input sequences and copies the required weights immediately before computation of each layer. Unlike DDB, OWR does not require sophisticated synchronization or load-balancing control between copy and computation, which lowers implementation complexity. Both techniques additionally require only small cacheable buffers approximately equal to the weight size of one or two layers, so the memory capacity overhead is minimal compared to weight duplication.

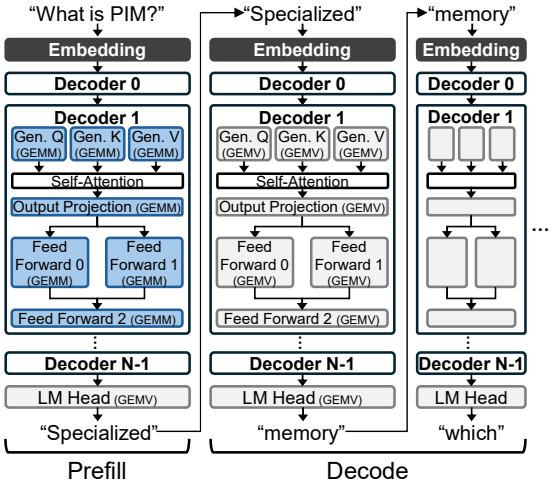


Figure 1: Model structure and inference process of LLM

The contributions of our study are as follows:

- We identify the inconsistency of the memory attribute between prefill and decode in PIM-enabled LLM inference, highlighting a fundamental system-level challenge unaddressed by prior works.
- We characterize the inconsistency in the weight layout between host-friendly and PIM-aware models, and quantify how naïve duplication inflates DRAM capacity on LLMs.
- We propose two software-only solutions, DDB and OWR, that employ small cacheable buffers and introduce a swizzled memory copy to resolve these inconsistencies without requiring any hardware modifications.

2 Background

2.1 Large language model inference

LLMs are built on a stack of transformer decoder layers [40]. Figure 1 shows the model structure of Llama 3.2 [27], a representative on-device LLM. The models process embedded input tokens sequentially to compute the probability distribution for the next token. Each decoder layer consists of query, key, and value projection layers, self-attention, and feed forward networks, typically featuring layer normalization within its sub-layers. LLM inference is divided into two distinct phases: the prefill phase, which handles long input sequences in a single pass and is dominated by compute-intensive GEMM (blue), and the subsequent decode phase, which processes one token per step and is constrained by memory bandwidth due to GEMV (gray) [8].

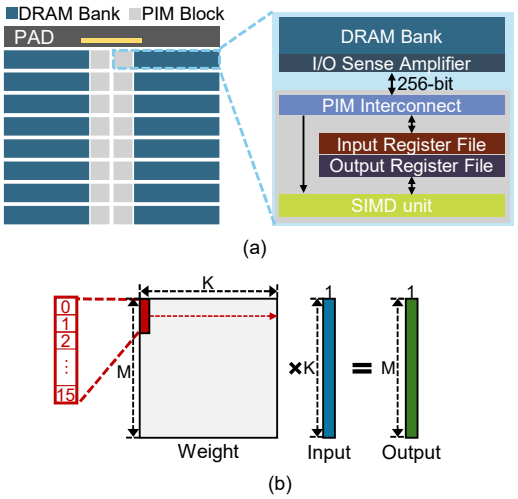


Figure 2: (a) LPDDR-PIM architecture, and (b) GEMV operating sequence of LPDDR-PIM

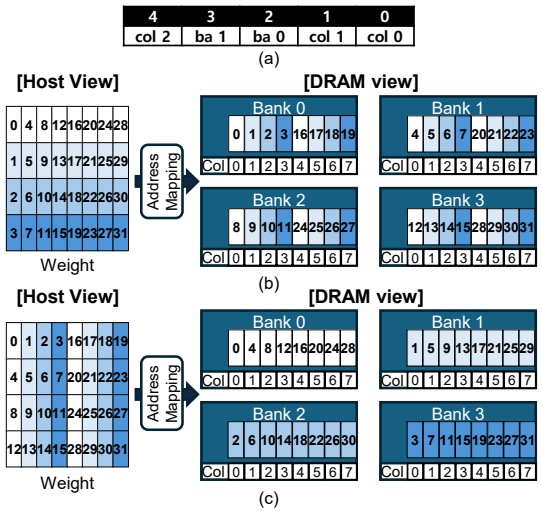


Figure 3: Mapping of host-friendly and PIM-aware weights inside DRAM. (a) Example address map. (b) Host and DRAM view of host-friendly weight and (c) PIM-aware weight where each bank holds a complete matrix row. Each rectangular block represents 16 weight elements in FP16/BF16 data format, which are stored in column-major order.

2.2 LPDDR-PIM architecture

LPDDR-PIM [18] and PIMnast [13] embed a SIMD unit inside each PIM block so that a 256-bit burst fetched from a DRAM bank can be processed in a single step. As shown in Figure 2(a), an LPDDR rank consists of 16 banks. Each PIM block connects to one or two banks and contains a SIMD unit alongside input/output register files. In multi-bank PIM mode, each SIMD unit performs 256-bit operations using inputs from the input register file and weights from a DRAM bank, and writes results back to output register file.

There are various data placement methodologies for performing GEMV operations using PIM. To determine the most suitable data placement for LPDDR-PIM, PIMnast [13] conducts an extensive design space exploration. The study reveals that storing weight elements in a column-major format per 256-bit DRAM request unit, depicted by the red rectangle in Figure 2(b), is optimal for GEMV acceleration. For example, in the FP16/BF16 data format, 16 weight elements are arranged in a column-major layout within a single DRAM request. During PIM operations, the 16 weight elements in the red rectangle compute with a single input element to produce 16 output partial sums. As shown by the red arrow, these partial sums are then continuously accumulated along the K dimension to yield the final output. We adopt the data placement strategy proposed by PIMnast.

2.3 Weight rearrangement for PIM

To execute LLMs on a PIM architecture, the weight matrix layout of conventional models must be rearranged to match the PIM execution characteristics. Figure 3 illustrates how the weights of a conventional model and a PIM-aware model are stored in DRAM. The address map is simplified to one channel, one rank, one bank group, four banks, one DRAM row, and eight DRAM columns (see Figure 3(a)). Figure 3(b) shows the conventional host-friendly layout. The host issues sequential addresses along the column-major direction of the weight matrix. To maximize DRAM bandwidth, the weights of sequential addresses are interleaved across multiple channels and banks. In contrast, PIM prefers that an entire matrix row reside within a single bank. As illustrated in Figure 3(c), mapping full rows to individual DRAM banks necessitates a PIM-aware rearrangement of the weights. As discussed in Section 2.2, the weights are organized to facilitate the continuous accumulation of partial sums along the K dimension to produce the final output inside each DRAM bank.

Algorithm 1 HBM-PIM GEMV pseudo code

```
for o = 0 to NumOutTile − 1 do
  for i = 0 to NumInputTile − 1 do
    Write8(InputAddr, InRfAddr)
    Offset ← AddrGen(ch, ra, bg, ba, ro, col)
    WeightAddr ← WeightAddr + Offset
    for r = 0 to NumTileRow − 1 do
      Read32(WeightAddr)
      WeightAddr ← WeightAddr + RowOffset
    end for
    InputAddr ← InputAddr + InTileOffset
  end for
  Read5(DummyAddr)
  Write8(OutRfAddr, OutputAddr)
  OutputAddr ← OutputAddr + OutTileOffset
end for
```

2.4 PIM execution

The host controls HBM-PIM [23] through conventional DRAM read and write requests [32]. Algorithm 1 illustrates an example of GEMV execution on HBM-PIM as pseudo code. Inputs are first written to the input register file via DRAM writes. The host then issues

DRAM reads to trigger MAC operations. Because HBM provides 32 DRAM columns per row, reads are grouped to units of 32 to maximize row-buffer hits. After completing MACs for the entire weight matrix, the host issues additional DRAM reads as NOPs to drain the SIMD ALU pipeline. The results in the output register file are then written back to DRAM.

For correct PIM operation, weights must be mapped to a non-cacheable region. During the MAC phase (*Read32*), each DRAM read corresponds to one MAC operation. If weights are placed in a cacheable region and remain in cache from prior accesses, subsequent reads are served by the cache rather than reaching the memory controller, preventing the PIM execution. Moreover, hardware prefetchers or cache locality policies may generate unintended DRAM requests, disrupting synchronization. Therefore, to ensure deterministic execution, all weights must be allocated in the non-cacheable region.

3 Motivation

3.1 Inconsistency challenges

The prefill and decode phases require different memory attributes and weight layouts for LLM weights.

Prefill phase (GEMM): Prefill is compute-intensive and typically executes on the host processor with multiple input tokens. During computation, repeated accesses to each weight element by up to the input sequence length (SL) times, lead to high weight reuse opportunity. Therefore, mapping the weight to a cacheable region is proven to be effective because it allows the host to reuse cached data. From a layout perspective, the host prefers a contiguous host-friendly layout, such as column-major or row-major order in the virtual address space.

Decode phase (GEMV): Decode is memory bandwidth-intensive and benefits from PIM when processing a single input token at a time. PIM can exploit up to an eightfold increase in effective DRAM bandwidth during the GEMV operation. PIM is triggered by DRAM requests which are issued by the host. If the weight is mapped to the cacheable region, DRAM requests can hit in cache and cannot reach the memory controller, which prevents the intended PIM operation. Hence, the weight must be mapped to a non-cacheable region. From a layout perspective, PIM requires a PIM-aware layout in which matrix data are contiguous within each DRAM bank. With this layout, each bank can locally generate a complete portion of the output.

Most prior studies on accelerating LLM inference with PIM do not explicitly consider the memory attribute and weight layout inconsistencies between the prefill and decode phases (see Table 1). IANUS [36], AttAcc [28], NeuPIMs [10], and AiM [20] primarily focus on architectural optimizations, KV-cache acceleration, and scheduling techniques. An exception is that HBM-PIM [23] and PAISE [21] circumvent the memory attribute inconsistency by placing host-friendly weights for prefill in a cacheable region and PIM-aware weights for decode in a non-cacheable region [31, 32]. However, such weight duplication introduces substantial DRAM capacity overhead.

Table 1: PIM related works to accelerate LLM

Works	Memory attribute	Weight layout	SW-only solution
IANUS	×	×	×
AttAcc	×	×	×
NeuPIMs	×	×	×
AiM	×	×	×
HBM-PIM	△	△	×
PAISE	△	△	○
FACIL	×	○	×
PIM-SHERPA	○	○	○

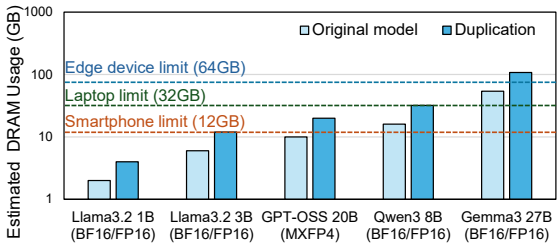


Figure 4: Estimated DRAM usage before and after weight duplication across representative LLMs. Y-axes use a logarithmic scale.

3.2 Current solutions and limitations

Weight duplication: A straightforward approach to maintain performance across both phases is to keep two separate copies of the model weights, each tailored to the specific memory attributes and access patterns of the corresponding phase. However, weight duplication worsens DRAM capacity constraints, particularly in LPDDR-based mobile and edge environments. Figure 4 shows the estimated DRAM usage under duplication. Llama 3.2 3B requires at least 6 GB of DRAM capacity. Because modern smartphones typically provide 8 - 12 GB DRAM capacity, smartphones can execute the Llama 3.2 3B model without duplication. However, enabling PIM requires an additional 6 GB for a second copy, which makes deployment infeasible on smartphones. Similarly, Qwen3 8B and Gemma3 27B can run on laptops or edge devices without duplication, but they become infeasible once PIM is enabled. As a result, adopting PIM may force the use of smaller, lower accuracy models.

Weight rearrangement: Without duplication, both host and PIM can be supported by converting the PIM-aware weight layout to a host-friendly layout at runtime, immediately before GEMM. However, online weight rearrangement can incur significant latency overhead due to additional resource usage. FACIL [37] reports that, for relatively short input SLs, online rearrangement increases the time-to-first-token (TTFT) by a factor of three.

To mitigate this overhead, FACIL augments the memory controller with lightweight address translation logic, which flexibly maps physical to DRAM addresses. This eliminates the need for weight duplication and online rearrangement. FACIL maintains a single PIM-aware copy of the weights. During the prefill phase, the

Table 2: Latency modeling across input sequence lengths (Samsung Galaxy S24+ CPU, FLOP/B ≈ 4)

Input	GEMM	DRAM	Online	SUM(GEMM, Online)		MAX(GEMM, Online)	
				time	vs GEMM	time	vs GEMM
1 - 4	t	t	$3t$	$4t$	400%	$3t$	300%
8	$2t$	t	$3t$	$5t$	250%	$3t$	150%
16	$4t$	t	$3t$	$7t$	175%	$4t$	100%
32	$8t$	t	$3t$	$11t$	138%	$8t$	100%
64	$16t$	t	$3t$	$19t$	119%	$16t$	100%
128	$32t$	t	$3t$	$35t$	109%	$32t$	100%
192	$48t$	t	$3t$	$51t$	106%	$48t$	100%

added logic presents the PIM-aware layout to the host in a host-friendly form, thereby preserving the GEMM performance on the host. However, the approach assumes a modified memory controller, which limits applicability in systems where the controller is fixed or cannot be changed. In addition, FACIL does not consider the inconsistency in the memory attribute between phases, which remains a fundamental challenge for PIM-enabled LLM deployment.

3.3 Revisiting online weight rearrangement overheads

FACIL deems online weight rearrangement impractical due to its substantial latency overhead. However, as input SL increases, the static cost of the online rearrangement is expected to account for a smaller fraction of the end-to-end latency. Notably, interactive LLMs [14] and on-device RAG [29, 35] continuously enlarge the input SL by appending previous responses or retrieved contexts to user queries. Motivated by these trends, this work revisits the conditions under which online weight rearrangement becomes practical and quantifies its overhead as a function of input SL.

The overhead of online weight rearrangement depends on the arithmetic intensity ($FLOP/B$) and the input SL. Table 2 models the overhead on a Samsung Galaxy S24+ device and shows how the overhead scales with varying input SL. For small inputs ($SL < 4$), GEMM latency is normalized to a base time (t). As input SL increases beyond four, latency grows roughly proportionally ($(input\ SL/(FLOP/B)) \times t$). The cost of online weight rearrangement is modeled as three DRAM transactions ($3t$): a read from the non-cacheable weight, a read of the destination from DRAM to cache, and a write to DRAM. $SUM(GEMM, Online)$ denotes serial execution, whereas $MAX(GEMM, Online)$ denotes overlapped execution. For input SL 1 - 8, fixed rearrangement cost dominates, resulting in 1.5 - 4× higher latency than GEMM alone. For input SL 16 - 192, growing GEMM time allows hiding rearrangement under computation as $MAX(GEMM, Online)$ indicates. Above input SL 192, GEMM dominates, so the rearrangement overhead shrinks. These observations indicate that for real workloads with sufficiently large input SL, the relative cost of online weight rearrangement can be amortized through the overlap with GEMM. Motivated by this insight, we propose PIM-SHERPA, a software-only method optimized for large input SL.

4 Contribution

To address inconsistencies in the memory attribute and weight layout between the prefill and decode phases, we introduce PIM-SHERPA, a software-only method for on-device LLM inference. The key idea is to allocate small DRAM buffers in the cacheable region and dynamically relocate weights from the non-cacheable region into these buffers during the prefill phase.

4.1 DRAM double buffering

To hide the latency of the online weight rearrangement, we first employ DRAM double buffering (DDB), which overlaps data movement with computation through a double buffering scheme. The cacheable DRAM buffer size is set to twice the weight size of the feed forward layer, which is typically the largest weight in a decoder with a uniform, repeated structure. During the prefill phase, while one cacheable buffer (buffer 0) is used to execute GEMM for the current layer, the weights for the next layer are prefetched from the non-cacheable region into the other buffer (buffer 1). The buffers alternate across successive decoder layers, allowing computation and data transfer to proceed in parallel. The proposed DDB eliminates the memory attribute inconsistency and performs weight rearrangement at runtime without requiring offline model duplication.

Figure 5 illustrates how DDB is applied to the Llama 3.2 structure. The generating query (Q) and output projection (O) layers have size $H{\times}H$. Because Llama 3.2 employs grouped-query attention [1], the generating key (K) and value (V) layers are one quarter the size of the Q layer. The feed forward (FF) layer is commonly four times larger than the Q layer, although recent models may deviate from this ratio. The cacheable DRAM buffer size is set to $2{\times}H{\times}4H$, which is twice the FF layer. All weight parameters reside in the non-cacheable region.

Before prefill begins, a preload step copies the GEMM weights into cacheable buffer 0. Because buffer 0 has size $H{\times}4H$, it can hold the weights of the Q, K, V, and O layers. To balance computation and transfer overhead, weights of these four layers are grouped and stored in buffer 0. When GEMM operations for Q, K, and V layers are executed, the weight of FF0 layer is copied into buffer 1 at the same time. This divides the weight of FF0 into four equal parts for load balancing so that one quarter is transferred during each of the three layers. Because DDB hides memory copies under

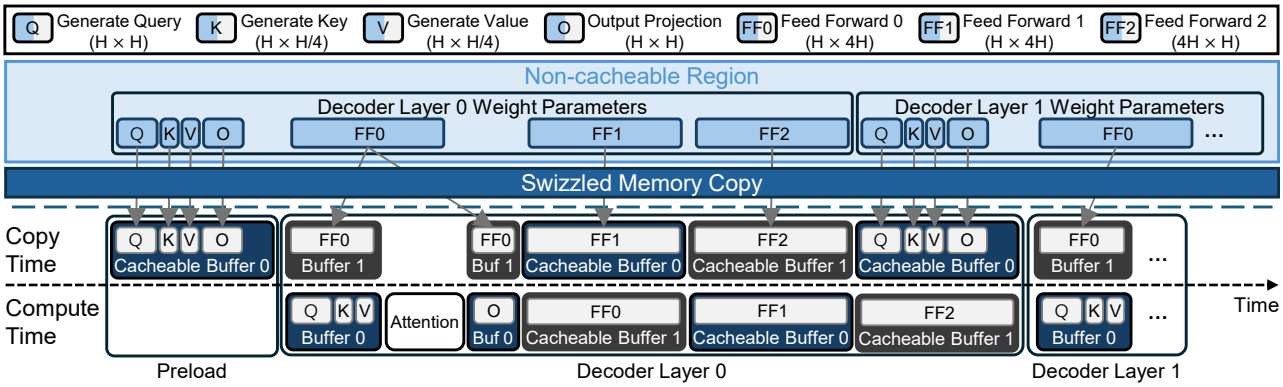


Figure 5: Illustration of DRAM double buffering mechanism during the prefill phase. Two small DRAM buffers are allocated within the cacheable region to prefetch weights of the subsequent layer from the non-cacheable region while GEMM is executed for the current layer. This double buffering approach effectively hides the latency of online rearrangement behind compute time. Each decoder layer (Q, K, V, O, FF0 (gate), FF1 (up), and FF2 (down)) corresponds to the Llama 3.2 structure where H represents the hidden dimension.

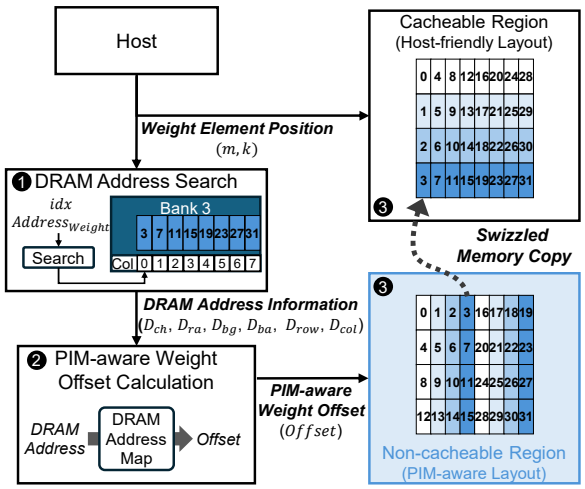


Figure 6: Swizzled memory copy process

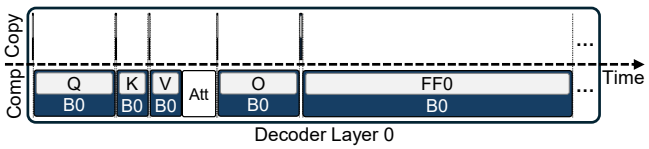


Figure 7: Illustration of online weight rearrangement with swizzled memory copy

compute, swizzled memory copies are not issued during relatively memory bound layers such as non-linear functions, normalization, and attention. As shown in the figure, the remaining quarter of the FF0 copy is completed during the O layer. FF0, FF1, and FF2 layers have a weight size equal to the DRAM buffer capacity and therefore fully occupy the buffer. During FF0 and FF1 layers, the amount of weight copied matches the amount consumed by the ongoing computation. FF2 layer performs a preload for the next decoder layer by preparing the weights of the Q, K, V, and O layers, which minimizes pipeline bubbles. The online weight rearrangement time of FF2 layer is the smallest because the K and V layers are one-quarter the size of the Q layer and the total weight size of Q, K, V, and O is *H*×*2.5H*.

4.2 Swizzled memory copy

Swizzled memory copy (SMC) is a key technique for resolving both memory attribute and weight layout inconsistencies. There are two approaches: 1) maintain the PIM-aware layout while copying from the non-cacheable to the cacheable region and incorporate swizzling logic into the GEMM kernel, or 2) convert the weights into a host-friendly layout during the copy so that the standard GEMM kernel can execute without modification. The first approach requires backend modifications and is hard to deploy with closed-source libraries, whereas the second confines the conversion to the copy path and leaves the backend untouched. Prioritizing portability and ease of integration, we adopt the second option and perform GEMM immediately after SMC.

Figure 6 illustrates the SMC process. Because memory addresses are managed in a one-dimensional (1D) space, the two-dimensional (2D) matrix coordinates (m, k) are converted into a 1D index (idx). Here, idx represents the logical position of a sequence of weight elements that are stored contiguously within a DRAM column. $Size_{col}$ denotes the size of the weight sequence stored in a DRAM column in bytes. $Element_{col}$ is $Size_{col}/Size_{element}$, where $Size_{element}$ denotes the byte size of a weight element. Since we employ a column-major layout, idx is calculated as $idx = \lfloor m/Element_{col} \rfloor + (M/Element_{col}) \times k$, where M denotes the number of rows in the matrix. Given the starting address of the weight, $Address_{WeightStart}$, the address of the corresponding weight sequence is calculated as Equation (1).

$$Address_{Weight} = Address_{WeightStart} + idx \times Size_{col} \quad (1)$$

❶ The information for each DRAM component is obtained using Equation (2). In this equation, idx and $Address_{Weight}$ are used to determine DRAM address fields such as D_{ch}, D_{ra}, D_{bg}, D_{ba}, D_{row}, and D_{col}. N denotes the sizes of the DRAM components.

$$\begin{cases} D_{ch} = idx \bmod N_{ch}, \\ D_{ra} = \left\lfloor \frac{idx}{N_{ch}} \right\rfloor \bmod N_{ra}, \\ D_{bg} = \left\lfloor \frac{idx}{N_{ch}N_{ra}} \right\rfloor \bmod N_{bg}, \\ D_{ba} = \left\lfloor \frac{idx}{N_{ch}N_{ra}N_{bg}} \right\rfloor \bmod N_{ba}, \\ D_{row} = \left\lfloor \frac{Address_{Weight}}{N_{ch}N_{ra}N_{bg}N_{ba}N_{col}Size_{col}} \right\rfloor, \\ D_{col} = \left\lfloor \frac{idx}{N_{ch}N_{ra}N_{bg}N_{ba}} \right\rfloor \bmod N_{col} \end{cases} \quad (2)$$

While the addresses for channels, ranks, bank groups, banks, and DRAM columns are periodically assigned via modulo operations using the relative sequence index idx, D_{row} specifies the absolute DRAM row where the weights of a target layer are mapped. Therefore, much like $Address_{WeightStart}$ serves as the base address in the host view, D_{row} relies on the absolute byte address ($Address_{Weight}$) rather than idx to correctly reflect the DRAM row offset allocated for each layer. ❷ The PIM-aware weight offset ($Offset$) is constructed according to the DRAM address mapping supported by the memory controller. Specifically, each DRAM address field is transformed and combined using bitwise XOR operations to generate $Offset$. ❸ Finally, the weight elements of the non-cacheable region are copied to the cacheable buffer. The $Weight_{Non-Cacheable}[Offset]$ is copied to the corresponding position $Buffer_{Cacheable}[idx]$.

4.3 Online rearrangement with swizzled memory copy

PIM-SHERPA also offers online weight rearrangement with SMC (OWR), which performs the swizzled memory copy serially just before GEMM when the input SL is sufficiently long and GEMM time dominates. OWR uses a single cacheable buffer that holds only one layer of weights required by the current computation step. The buffer size is configured as $H{\times}4H$, corresponding to the FF layer weight size, which is typically the largest layer in the model.

As illustrated in Figure 7, once required weights are copied into the buffer, OWR triggers GEMM using the rearranged data. This serialized procedure ensures a deterministic execution order and uniform memory access behavior across layers. This design can improve software simplicity by eliminating the need for complex interleaving or sophisticated synchronization between SMC and compute phases. OWR is particularly effective in long sequence scenarios where computation dominates overall performance. Because the SMC latency remains static regardless of the input SL, its relative impact on total runtime diminishes as the GEMM time increases.

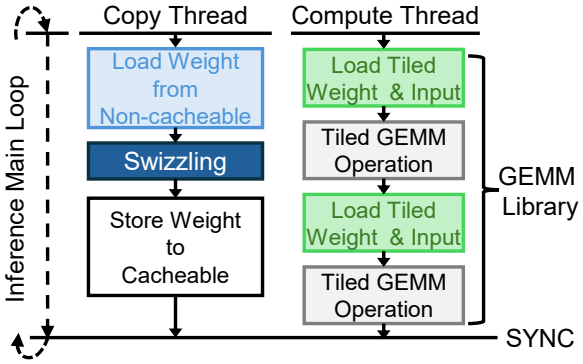


Figure 8: DRAM double buffering runtime pipeline

4.4 Implementation details

We discuss the implementation details of DDB and OWR. Our base system is ExecuTorch [6], a modern runtime for on-device AI inference that supports recent LLMs, such as Llama, on smartphone CPUs. By leveraging this system, our implementation aims to maximize hardware resource utilization while adhering to strict memory and computation constraints of mobile devices.

Figure 8 illustrates the implementation details of DDB. The system employs dedicated compute and copy threads which are managed by partitioning the available CPU resources and binding each thread to a specific core through thread pinning. The copy threads run SMC, which loads weights from the non-cacheable region, swizzles the PIM-aware layout into a host-friendly layout, and stores the results in the cacheable region. Concurrently, the compute threads execute GEMM with the optimized library supplied by the framework, without modification. To effectively hide rearrangement latency, we employ a double-buffering scheme; the compute threads read tiled weights from one cacheable buffer and compute GEMM operations, while the copy threads write to the other cacheable buffer, thereby overlapping data movement with computation. DDB synchronizes these two thread groups at the boundary of each sub-layer.

When it comes to OWR, the copy threads first perform SMC into a single buffer, and then the compute threads run GEMM on the prepared weights. Unlike DDB, OWR does not require complex pipeline synchronization between copy and compute phases, which reduces the implementation complexity.

5 Experimental Setup

We evaluated PIM-SHERPA using ExecuTorch [6]. We used a Samsung Galaxy S24+ and BF16 data format, as the ExecuTorch officially supported to run Llama models on the Galaxy S24+ device. The Samsung Galaxy S24+ is equipped with Exynos 2400 (see Table 3) and its CPU system consists of a single big core (Cortex-X4), two middle-high and three middle-low cores (A720), and four little cores (A520). To utilize Neon SIMD architecture, we only used the big and middle cores.

The six CPU cores together achieve a peak performance of 321 GFLOPS. The system has LPDDR5X-8533 with four channels, providing up to 68.264 GB/s. Arithmetic intensity (FLOP/B) is 4.7. We

Table 3: Host and PIM emulation system specification

Section	Field	Value
Host System	Device	Samsung Galaxy S24+
	Host CPU	1 × Cortex-X4 2 × A720 @ 2.9 GHz 3 × A720 @ 2.7 GHz
	BF16 GFLOPS	321
	DRAM BW	68.264 GB/s
PIM Emulation System	Device	LPDDR5X-PIM
	Input RF	8
	Output RF	8
	ALU precision	BF16

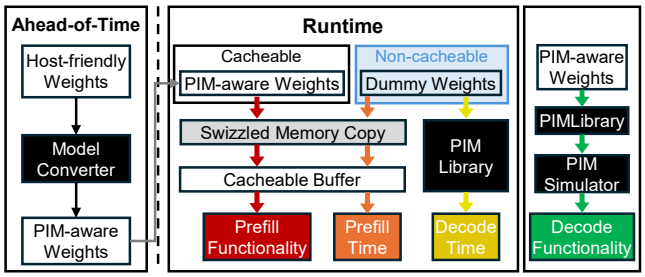


Figure 9: Prefill-decode execution time and functionality evaluation pipeline. Red and green indicate functionality paths for prefill and decode, respectively. Orange and yellow indicate execution time paths. Black boxes denote supported software.

used the Llama 3.2 1B and 3B models released as an on-device model by META [27]. The hidden sizes for Llama 3.2 1B and 3B models are 2048 and 3072, respectively. The batch size was set to 1. We assigned four CPU cores to GEMM because using six cores introduces straggle threads due to the asymmetric performance between big and middle core, which forced the faster cores to idle during synchronization barriers, yielding performance similar to achieved with four cores. Two cores were allocated for copy threads.

We describe the execution time and functionality paths as shown in Figure 9 in detail. We used a model converter to transform host-friendly weights into PIM-aware weights based on the address mapping of the Galaxy S24+. The PIM execution requires the placement of the weights in a single physically contiguous memory region because weight element addresses are calculated by combining base address of each layer with specific offset information. In Linux, such large physically contiguous allocations are typically managed by the contiguous memory allocator (CMA). Furthermore, to ensure that PIM operations are reliably triggered, we utilized this contiguous region as a non-cacheable region.

Prefill time emulation (orange): SMC converts PIM-aware weights from a non-cacheable region and transfers the resulting host-friendly weights to a cacheable buffer. In parallel, the host performs GEMM using the host-friendly weights stored in the cacheable buffer. We measured the prefill time while SMC and GEMM run concurrently. On the Galaxy S24+, the CMA-backed contiguous region that can be configured as non-cacheable is approximately 1 GB, while the Llama 3.2 1B and 3B models require at least 2.4 GB and 6 GB of weights, respectively. The device cannot accommodate all weights within the available non-cacheable contiguous region. Therefore, we used dummy weights for prefill time emulation. If the full LLM weights could fit in the CMA-backed contiguous region, the same PIM-aware weights could be used for both execution time and functionality paths, enabling measurement of actual execution time on the device rather than emulation.

Table 4: GEMV execution time comparison between HBM-PIM and HBM-PIM emulation

Output Dim	Input Dim	HBM-PIM Relative Time	Emulation Relative Time	Emulation Error
4096	2048	1	0.965	3.54%
	4096	1	0.980	2.02%
	8192	1	0.969	3.05%
	16384	1	0.993	0.66%
8192	2048	1	0.990	0.97%
	4096	1	0.993	0.69%
	8192	1	0.999	0.14%
	16384	1	0.996	0.35%

Prefill functionality validation (red): To verify functionality, we needed to use real PIM-aware weights. However, as previously mentioned, entire weights cannot be stored in the non-cacheable region. Thus, we placed the weights into the cacheable region and performed SMC, where PIM-aware weights were converted into a host-friendly layout within the cacheable buffer. We then performed GEMM using this layout and validated the functionality by comparing the prefill outputs of PIM-SHERPA against those of the original ExecuTorch.

Decode time emulation (yellow): We emulated a PIM system using LPDDR5X and PIMLibrary [32]. We modified PIMLibrary to support ARM CPUs and emulated LPDDR5X-PIM performance. To ensure that every PIM request reaches the memory controller, we used the dummy weights in the non-cacheable region. We measured the execution time of LPDDR5X-PIM using PIMLibrary with the dummy weights.

To substantiate the accuracy of our PIM emulation, we compare real HBM-PIM hardware against HBM-PIM emulation and observe a small performance gap. Table 4 reports GEMV execution times measured on real HBM-PIM hardware and on HBM-PIM emulation. Across multiple input and output dimensions, the emulation error remains low (0.1% - 3.6%) relative to real HBM-PIM. The small gap can be explained as follows. HBM-PIM does not introduce additional PIM-specific requests; instead, it operates using conventional DRAM read and write requests. Moreover, HBM-PIM uses the same memory controller and timing parameters as conventional HBM. Therefore, when we issue the same DRAM request pattern that triggers HBM-PIM GEMV execution on an AMD MI100 GPU system equipped with conventional HBM, the externally observed memory access and service time closely matches that of real HBM-PIM. We

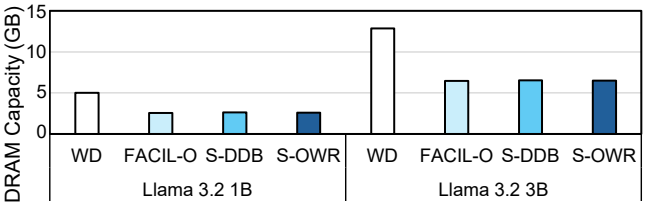


Figure 10: Required DRAM capacity across WD, FACIL-O, S-DDB and S-OWR

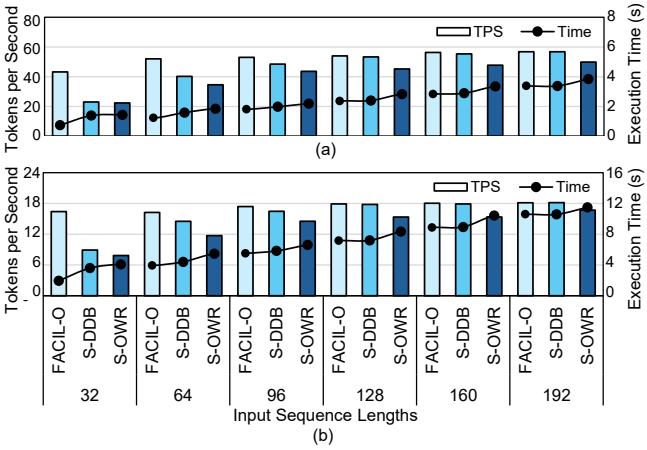


Figure 11: TTFT across input sequence lengths using Llama 3.2 (a) 1B and (b) 3B models. Results are presented by tokens per second (TPS) and execution time. For TTFT, WD yields performance identical to FACIL-O.

incorporated LPDDR5X-PIM parameters into the HBM-PIM emulation system to develop a high-fidelity LPDDR5X-PIM emulation system, which has undergone industry-level validation.

Decode functionality validation (green): Without an actual PIM device, we cannot directly verify whether the emulation produces correct GEMV results. To address this, we validate functionality by coupling PIMLibrary with a PIM Simulator [33]. The simulator performs GEMV using real input and weight values. For validation, we stored LLM weights in the simulator DRAM at the same DRAM addresses used on the mobile device. We then fed the DRAM requests generated by PIMLibrary into the simulator and compare the resulting GEMV outputs against those computed by the host CPU.

6 Evaluation

We present evaluation results by applying PIM-SHERPA to Llama 3.2 inference. To understand the results in detail, we consider four scenarios: weight duplication (WD), FACIL under an oracle assumption (FACIL-O), SHERPA-DDB (S-DDB), and SHERPA-OWR (S-OWR). In WD, host-friendly weights are stored in a cacheable region, whereas PIM-aware weights are placed in a non-cacheable region. To compare FACIL with PIM-SHERPA, we assume that FACIL resolves a memory attribute inconsistency; we denote this as FACIL-O.

DRAM capacity saving: S-DDB and S-OWR markedly reduce the required DRAM capacity as shown in Figure 10. For the Llama 3.2 1B BF16 model, host-friendly weights require 2.47 GB. To avoid misalignment, PIM-aware weights include additional padding in the K and V layers and in the LM head. This padding increases size by 80 MB relative to the host-friendly baseline. In WD, both host-friendly and PIM-aware weights are stored, whereas S-DDB and S-OWR only require PIM-aware weights and small additional cacheable buffers. This results in memory savings of 47.8% and 48.5% relative to WD. The cacheable buffer size is 32 MB (2048×8192×2 B). Because S-OWR uses a single cacheable buffer, its required capacity is about 0.7% smaller than S-DDB. FACIL-O augments the memory controller and keeps only PIM-aware weights, reducing the required capacity by 49.2%. The Llama 3.2 3B BF16 model requires 6.4 GB for host-friendly weights, and S-DDB and S-OWR save 49.4% and 49.7% of DRAM capacity, respectively. With software-only support, S-OWR and S-DDB achieve significant capacity savings comparable to FACIL-O while resolving the memory attribute and weight layout inconsistencies.

TTFT performance: The TTFT of PIM-SHERPA is nearly identical to, or slightly lower than, that of FACIL-O as the input SL increases. Figure 11 shows that S-DDB achieves TTFT comparable to FACIL-O for the input SL between 128 and 192. S-DDB uses two copy threads, and the static SMC latency from the non-cacheable region to the cacheable region is about 0.89 and 2.54 s on Llama 3.2 1B and 3B models, respectively. When the GEMM compute time exceeds this latency (SL ≥ 128), S-DDB effectively hides the SMC latency. In addition, as the input SL increases, the performance of S-OWR gradually approaches that of the others. At 192-input SL on the 3B model, S-OWR achieves 16.7 TPS, nearly matching FACIL-O and S-DDB. S-OWR performs the SMC and GEMM sequentially using four threads. The measured SMC latencies of S-OWR are approximately 0.6 and 1.4 s on Llama 3.2 1B and 3B models, respectively. As a result, S-OWR always incurs SMC latency and lags behind the others.

The overlap achieved by S-DDB differs from the theoretical estimate in Section 3.3. Table 2 indicates that the SMC latency of S-DDB should overlap with GEMM time at the 16-input SL. However, we found that data movement from the non-cacheable to the cacheable regions is about twice as slow as movement between cacheable regions through experiments. In addition, using only two copy threads does not saturate available memory bandwidth. As a result, SMC achieves only about one quarter of the peak DRAM bandwidth. At 32-input SL on the 1B model, S-DDB and S-OWR exhibit similar latencies: S-OWR combines sums 0.74 s of GEMM time with 0.6 s of SMC latency on four threads, whereas S-DDB incurs about 0.89 s with two copy threads plus additional contention overhead.

S-DDB analysis: Figure 12 shows the per-thread timeline within a decoder layer. The copy threads perform SMC once for PIM-aware weights regardless of input SL, resulting in a constant SMC latency. In contrast, GEMM time scales linearly with input SL. At 64-input SL, the copy thread becomes the critical path for most layers (Figure 12(a)). Because K and V layers use grouped-query attention, their matrices are smaller, which makes their compute

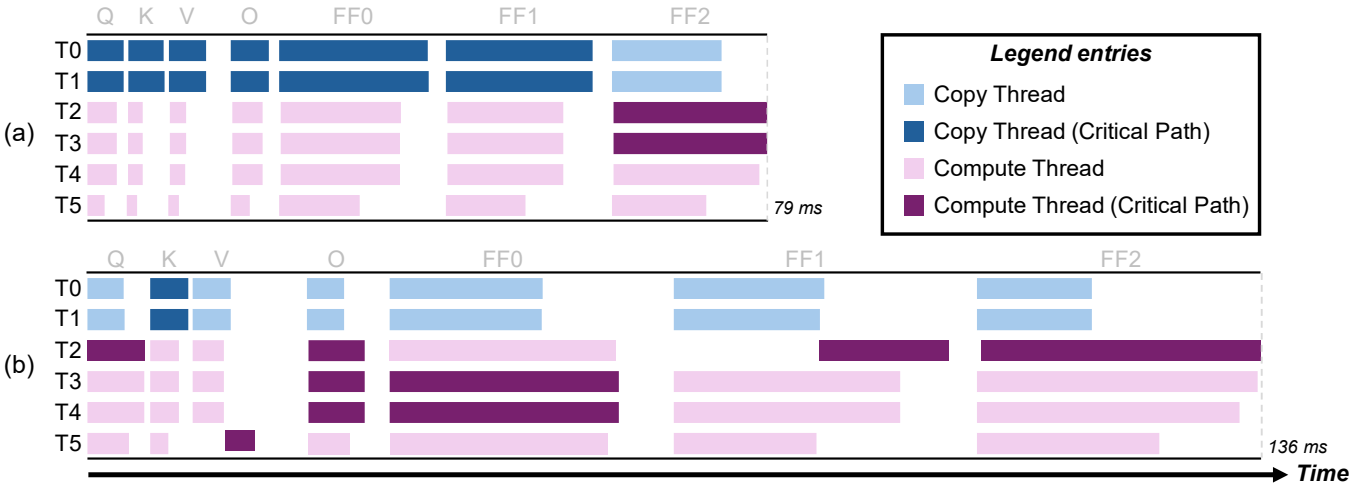


Figure 12: S-DDB execution timeline of a single decoder layer at input sequence lengths of (a) 64 and (b) 128 tokens on the Galaxy S24+ using Llama 3.2 1B model. Blue bars denote the copy thread and purple bars indicate the compute thread. Each bold color bar highlights the critical path within each layer. Thread 5 is the main thread and is bound to the Cortex-X4. Threads 0 and 1 are pinned to the A720 cores with 2.9 GHz and threads 2 - 4 are pinned to the A720 cores with 2.7 GHz.

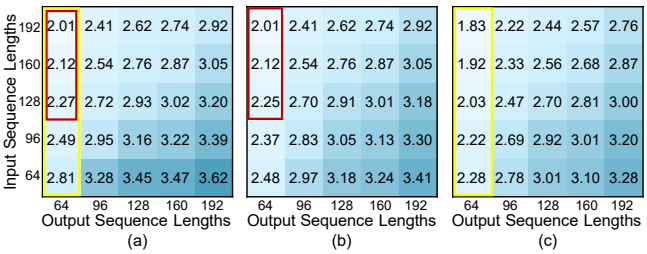


Figure 13: End-to-end inference speedup of (a) FACIL-O, (b) S-DDB, and (c) S-OWR over a host-only LPDDR5X system when using LPDDR5X-PIM. Input and output sequence lengths vary from 64 to 192 . We used Llama 3.2 1B model.

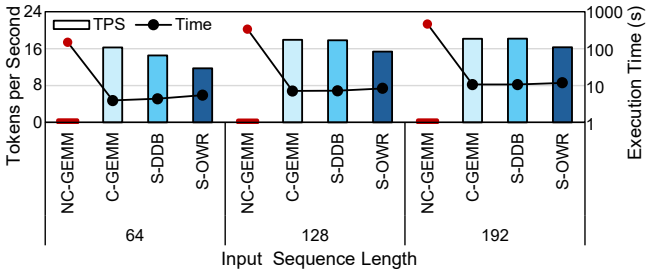


Figure 14: TTFT comparison when the device has only a single weight due to limited DRAM capacity. We used Llama 3.2 3B model. C-GEMM denotes GEMM with cacheable weight without PIM. NC-GEMM stands for GEMM with non-cacheable weights using PIM. Y-axes of execution time uses a logarithmic scale.

time much shorter than that of other layers. Because the K and V layers copy the same amount of FF0 weight as the Q and O layers, the gap between compute time and copy latency becomes pronounced. The FF2 layer prefetches the weights of the Q, K, V, and O layers, partially hiding the SMC latency because the weights of K and V layer are relatively small.

At 128-input SL, the SMC latency of all layers except K can be hidden to the compute latency (see Figure 12(b)). In the V layer, even though the compute time is shorter, the layer may complete later due to scheduling issues when hardware threads are unavailable. Because the number of SIMD units on a Cortex-X4 is twice that of an A720, T5 finishes GEMM earlier across all layers. In the FF1 layer, T2 fails to enqueue into the run queue due to limited scheduling slots and remains pending until T5 completes on the Coretex-X4. Once T5 completes, the scheduler dispatches T2 to the available core, resulting in delayed completion.

LLM inference speedup using PIM: S-DDB and FACIL-O exhibit similar end-to-end inference speedup with LPDDR5X-PIM across most combinations of input and output SLs (see Figure 13). For input SL with ≥ 128, the speedup of FACIL-O and S-DDB is nearly identical, even when output SL is relatively small (red box). As the input SL increases, the performance gap between S-OWR and the others gradually narrows (yellow box). For input SLs 64 - 96, S-DDB differs from FACIL-O by about 2.5% - 19%. Although S-OWR shows smaller gains, it still delivers up to 3.3× speedup over the non-PIM baseline.

TTFT performance on limited DRAM capacity: PIM-SHERPA remains useful even when the host can keep only a single copy of the weights due to limited DRAM capacity (see Figure 14). To reflect realistic device constraints, we evaluated the Llama-3.2 3B model on the Galaxy S24+, which can hold only one copy of the weights. C-GEMM executes GEMM using weights in the cacheable region. During prefill, C-GEMM can surpass S-DDB and S-OWR; however, once the decode phase is included, it is not a viable substitute for PIM. Because NC-GEMM cannot reuse cached weights, it rereads

weights from DRAM approximately input SL times. Across the evaluated range, the execution time of NC-GEMM grows roughly in proportion to input SL, reaching 148 s, 332 s, and 458 s at input SLs of 64, 128, and 192, respectively. Throughput remains about 0.4 TPS regardless of the input SL. These findings also confirm that the device correctly enforces the non-cacheable setting.

7 Discussion

Applicability to GPU and NPU systems: PIM-SHERPA can be applied to GPU systems that include a last-level cache. The GPU tiles GEMM at thread-block granularity to distribute work across streaming multiprocessors (SMs) [24]. The GPU uses the L2 cache to avoid repeatedly reading the same data from DRAM across SMs. On a GPU with PIM, PIM-SHERPA can reduce data movement by approximately the input SL divided by the thread-block tile size during prefill. From an implementation perspective, it can leverage a GPU software pipeline [22, 26], which hides DRAM to shared memory transfers under compute. Integrating SMC into the pipeline preserves the same latency hiding benefit.

The application to NPUs depends on the memory hierarchy. If an NPU includes both on-chip SRAM and a cache [4, 39], the approach mirrors the CPU and GPU cases. If an NPU provides only SRAM [16, 17], we can allocate the double buffer directly to SRAM. The system performs the swizzled memory copy from DRAM to SRAM, which completely eliminates the DRAM write-back overhead, making SMC even more efficient on SRAM-only NPUs. Thus, we emphasize the universality of our approach.

Online rearrangement overhead over FLOP/B: On systems with higher compute capability, the overhead of online rearrangement can appear larger. NVIDIA GPUs are representative of high FLOP/B platforms. NVIDIA Jetson AGX Orin (edge) and B100 (server) provide about 207 and 241 FLOP/B, respectively. With an input SL of 128, as used in our experiments, TTFT can increase up to 3 times compared to ideal compute time on such platforms. However, systems with high compute capability can also use the input SL more freely. Recent LLM work [11] reports effective input SLs up to 128K, and the growth in input SL can exceed the increase in FLOP/B by roughly an order of magnitude. Thus, the method remains applicable on high FLOP/B systems, although the preferred operating window in terms of FLOP/B and input SL may shift.

Scalability of cacheable DRAM buffer across LLMs: We analyze the requirements of DRAM buffer in a spectrum of modern LLMs, ranging from 1B to 27B parameters. As detailed in Section 4.1, the total size of cacheable DRAM buffers is required to be twice the weight of the FF layer, which is typically the largest weight in a uniform decoder. Because DRAM buffers are reused for the entire decoder stack, the buffer capacity is driven strictly by the hidden and FF dimensions, entirely independent of the number of decoder layers. Consequently, the required buffer size grows sub-linearly relative to the overall model size. For example, a smaller model like Llama 3.2 1B (2.4 GB) requires a 64 MB buffer, accounting for 2.7% of its total memory footprint. As model scale increases to Llama2 7B (12.6GB) and Qwen3 8B (14.4GB), the absolute buffer sizes increase to 172 MB and 192 MB, respectively, but their relative overheads drop to approximately 1.3%. For a larger model such as Gemma3 27B (53.1 GB), the required buffer is 441 MB, which represents 0.83% of the total model memory. This sub-linear scaling demonstrates that as models continue to grow, the relative memory footprint overhead of our design inherently diminishes, proving that PIM-SHERPA provides a highly scalable and future-proof solution for on-device inference under strict memory constraints.

8 Related Work

PIM software works: A body of research addresses PIM inefficiencies in software. SimplePIM [3] simplifies programming with an iterator-based API and communication primitives. PIM-STM [25] shows that transactional memory applications can be effectively accelerated on PIM. PIM-CARE [12] provides a compiler for concurrent multi-application execution and significantly improves PIM resource utilization. These studies use the commercial UPMEM device [5], which positions DRAM as an accelerator and does not consider inconsistencies in memory attribute and weight layout. PAISE [21] is a software study that uses Samsung HBM-PIM internally and improves PIM utilization with a data layout adjustment module. However, like HBM-PIM, it duplicates weights to avoid these inconsistencies.

LLM inference acceleration using PIM: Several studies accelerate LLM inference with PIM. IANUS [36] and NeuPIMs [10] orchestrate an NPU and PIM to execute layers with different characteristics in parallel. AttAcc [28] partitions and balances requests with heterogeneous sequence lengths to accelerate the KV cache. These works do not consider memory attribute and weight layout inconsistencies. HBM-PIM [23] and PAISE [21] circumvent these inconsistencies via weight duplication. FACIL [37] resolves the weight layout inconsistency through flexible address mapping but does not consider the memory attribute inconsistency.

9 Conclusion

To the best of our knowledge, this is the first work to explicitly focus on memory attribute inconsistency and propose effective solutions on PIM-enabled systems. PIM-SHERPA introduces two software-only techniques that resolve both memory attribute and weight layout inconsistencies at runtime. On a commercial on-device platform, the approach reduces the DRAM capacity usage compared to weight duplication approximately 47.8 - 48.5%. Our approach also delivers TTFT comparable to FACIL, while it still preserves PIM speedup. Our study suggests that addressing these challenges is crucial for the future commercialization of PIM.

References

[1] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. arXiv:2305.13245 [cs.CL] https://arxiv.org/abs/2305.13245

[2] Apple. 2024. Apple Intelligence. https://www.apple.com/apple-intelligence/.

[3] Jinfan Chen, Juan Gómez-Luna, Izzat El Hajj, Yuxin Guo, and Onur Mutlu. 2024. SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory. In *Proceedings of the 32nd International Conference on Parallel Architectures and Compilation Techniques* (Vienna, AE, Austria) *(PACT '23)*. IEEE Press, Piscataway, NJ, USA, 99–111. doi:10.1109/PACT58117.2023.00017

[4] Joel Coburn, Chunqiang Tang, Sameer Abu Asal, Neeraj Agrawal, Raviteja Chinta, Harish Dixit, Brian Dodds, Saritha Dwarakapuram, Amin Firoozshahian, Cao Gao, Kaustubh Gondkar, Tyler Graf, Junhan Hu, Jian Huang, Sterling Hughes, Adam Hutchin, Bhasker Jakka, Guoqiang Jerry Chen, Indu Kalyanaraman, Ashwin Kamath, Pankaj Kansal, Erum Kazi, Roman Levenstein, Mahesh Maddury, Alex Mastro, Siji Medaiyese, Pritesh Modi, Jack Montgomery, Satish Nadathur, Amit Nagpal, Ashwin Narasimha, Maxim Naumov, Eleanor Ozer, Jongsoo Park, Poorvaja

Ramani, Harikrishna Reddy, David Reiss, Deboleena Roy, Sathish Sekar, Arushi Sharma, Pavan Shetty, Aravind Sukumaran-Rajam, Eran Tal, Mike Tsai, Shreya Varshini, Richard Wareing, Olivia Wu, Xiaolong Xie, Jinghan Yang, Hangchen Yu, Tanmay Zargar, Zitong Zeng, Feixiong Zhang, Ajit Matthews, Xun Jiao, Jiyuan Zhang, Emmanuel Menage, Truls Edvard Stokke, and Mohammed Sourouri. 2025. Meta's Second Generation AI Chip: Model-Chip Co-Design and Productionization Experiences. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 1689–1702. doi:10.1145/3695053.3731409

[5] Fabrice Devaux. 2019. The true Processing In Memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE, Piscataway, NJ, USA, 1–24. doi:10.1109/HOTCHIPS.2019.8875680

[6] ExecuTorch. 2023. Executorch. https://github.com/pytorch/executorch.

[7] Google. 2016. Google Assistant. https://assistant.google.com/.

[8] Congjie He, Yeqi Huang, Pei Mu, Ziming Miao, Jilong Xue, Lingxiao Ma, Fan Yang, and Luo Mai. 2025. WaferLLM: Large Language Model Inference at Wafer Scale. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 257–273. https://www.usenix.org/conference/osdi25/presentation/he

[9] Siyuan He, Zhantong Zhu, Yandong He, and Tianyu Jia. 2025. LP-Spec: Leveraging LPDDR PIM for Efficient LLM Mobile Speculative Inference with Architecture-Dataflow Co-Optimization. arXiv:2508.07227 [cs.AR] https://arxiv.org/abs/2508.07227

[10] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. 2024. NeuPIMs: NPU-PIM Heterogeneous Acceleration for Batched LLM Inferencing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 722–737. doi:10.1145/3620666.3651380

[11] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. RULER: What's the Real Context Size of Your Long-Context Language Models? arXiv:2404.06654 [cs.CL] https://arxiv.org/abs/2404.06654

[12] Inyong Hwang, Donghyeon Kim, Seokwon Kang, Taehyeong Park, Taehoon Kim, Jiwon Seo, Hanjun Kim, Youngsok Kim, and Yongjun Park. 2025. PIM-CARE: A Compiler-Assisted Dynamic Resource Allocation Framework for Real-world DRAM PIM. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 458–472. doi:10.1145/3721145.3725777

[13] Mohamed Assem Ibrahim, Mahzabeen Islam, and Shaizeen Aga. 2025. PIMnast: Balanced Data Placement for GEMV Acceleration with Processing-In-Memory. In *Proceedings of the SC '24 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis* (Atlanta, GA, USA) *(SC-W '24)*. IEEE Press, Piscataway, NJ, USA, 970–981. doi:10.1109/SCW63240.2024.00137

[14] Jinwoo Jeong and Jeongseob Ahn. 2025. Accelerating LLM Serving for Multi-turn Dialogues with Efficient Resource Management. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3676641.3716245

[15] Kyoungho Jeun, Hyeonu Kim, and Eojin Lee. 2025. Fold-PIM: A Cost-Efficient LPDDR5-Based PIM for On-Device SLMs. *IEEE Computer Architecture Letters* 24, 1 (2025), 185–188. doi:10.1109/LCA.2025.3566692

[16] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the Graphcore IPU Architecture via Microbenchmarking. arXiv:1912.03413 [cs.DC] https://arxiv.org/abs/1912.03413

[17] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 82, 14 pages. doi:10.1145/3579371.3589350

[18] Byeongho Kim, Sanghoon Cha, Sangsoo Park, Jieun Lee, Sukhan Lee, Shin-haeng Kang, Jinin So, Kyungsoo Kim, Jin Jung, Jong-Geon Lee, Sunjung Lee, Yoonah Paik, Hyeonsu Kim, Jin-Seong Kim, Won-Jo Lee, Yuhwan Ro, YeonGon Cho, Jin Hyun Kim, JoonHo Song, Jaehoon Yu, Seungwon Lee, Jeonghyeon Cho, and Kyomin Sohn. 2024. The Breakthrough Memory Solutions for Improved Performance on LLM Inference. *IEEE Micro* 44, 3 (May 2024), 40–48. doi:10.1109/MM.2024.3375352

[19] Byeongho Kim, Jongwook Chung, Eojin Lee, Wonkyung Jung, Sunjung Lee, Jaewan Choi, Jaehyun Park, Minbok Wi, Sukhan Lee, and Jung Ho Ahn. 2020. MViD: Sparse Matrix-Vector Multiplication in Mobile DRAM for Accelerating Recurrent Neural Networks. *IEEE Trans. Comput.* 69, 7 (July 2020), 955–967. doi:10.1109/TC.2020.2984496

[20] Guhyun Kim, Jinkwon Kim, Nahsung Kim, Woojae Shin, Jongsoon Won, Hyunha Joo, Haerang Choi, Byeongju An, Gyeongcheol Shin, Dayeon Yun, Jeongbin Kim, Changhyun Kim, Ilkon Kim, Jaehan Park, Yosub Song, Byeongsu Yang, Hyeongdeok Lee, Seungyeong Park, Wonjun Lee, Seonghun Kim, Yonghoon Park, Yousub Jung, Gi-Ho Park, and Euicheol Lim. 2024. SK Hynix AI-Specific Computing Memory Solution: From AiM Device to Heterogeneous AiMX-xPU System for Comprehensive LLM Inference. In *2024 IEEE Hot Chips 36 Symposium (HCS)*. IEEE Press, Piscataway, NJ, USA, 1–26. doi:10.1109/HCS61935.2024.10664793

[21] Hyojung Lee, Daehyeon Baek, Jimyoung Son, Jieun Choi, Kihyo Moon, and Minsung Jang. 2025. PAISE: PIM-Accelerated Inference Scheduling Engine for Transformer-based LLM. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Press, Piscataway, NJ, USA, 1707–1719. doi:10.1109/HPCA61900.2025.00126

[22] Sunjung Lee, Seunghwan Hwang, Michael Jaemin Kim, Jaewan Choi, and Jung Ho Ahn. 2022. Future Scaling of Memory Hierarchy for Tensor Cores and Eliminating Redundant Shared Memory Traffic Using Inter-Warp Multicasting. *IEEE Trans. Comput.* 71, 12 (2022), 3115–3126. doi:10.1109/TC.2022.3207134

[23] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, New York, NY, USA, 43–56. doi:10.1109/ISCA52012.2021.00013

[24] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. 2019. A coordinated tiling and batching framework for efficient GEMM on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 229–241. doi:10.1145/3293883.3295734

[25] André Lopes, Daniel Castro, and Paolo Romano. 2024. PIM-STM: Software Transactional Memory for Processing-In-Memory Systems. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 897–911. doi:10.1145/3620665.3640428

[26] Sangkug Lym, Donghyuk Lee, Mike O'Connor, Niladrish Chatterjee, and Mattan Erez. 2019. DeLTA: GPU Performance Model for Deep Learning Applications with In-Depth Memory System Traffic Analysis. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Press, Piscataway, NJ, USA, 293–303. doi:10.1109/ISPASS.2019.00041

[27] Meta. 2024. *Llama 3.2: Revolutionizing edge AI and vision with open, customizable models.* Accessed 2025-10-26. https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/

[28] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn. 2024. AttAcc! Unleashing the Power of PIM for Batched Transformer-based Generative Model Inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 103–119. doi:10.1145/3620665.3640422

[29] Taehwan Park, Geonho Lee, and Min-Soo Kim. 2025. MobileRAG: A Fast, Memory-Efficient, and Energy-Efficient Method for On-Device RAG. arXiv:2507.01079 [cs.DB] https://arxiv.org/abs/2507.01079

[30] Qualcomm. 2024. Qualcomm AI hub. https://aihub.qualcomm.com/.

[31] Samsung. 2022. PIMAiCompiler. https://github.com/SAITPublic/PimAiCompiler

[32] Samsung. 2022. PIMLibrary. https://github.com/SAITPublic/PIMLibrary

[33] Samsung. 2022. *PIMSimulator*. https://github.com/SAITPublic/PIMSimulator

[34] Samsung. 2024. Galaxy AI. https://www.samsung.com/us/galaxy-ai/.

[35] Korakit Seemakhupt, Sihang Liu, and Samira Khan. 2024. EdgeRAG: Online-Indexed RAG for Edge Devices. arXiv:2412.21023 [cs.LG] https://arxiv.org/abs/2412.21023

[36] Minseok Seo, Xuan Truong Nguyen, Seok Joong Hwang, Yongkee Kwon, Guhyun Kim, Chanwook Park, Ilkon Kim, Jaehan Park, Jeongbin Kim, Woojae Shin, Jongsoon Won, Haerang Choi, Kyuyoung Kim, Daehan Kwon, Chunseok Jeong, Sangheon Lee, Yongseok Choi, Wooseok Byun, Seungcheol Baek, Hyuk-Jae Lee, and John Kim. 2024. IANUS: Integrated Accelerator based on NPU-PIM Unified Memory System. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 545–560. doi:10.1145/3620666.3651324

[37] Seong Hoon Seo, Junghoon Kim, Donghyun Lee, Seonah Yoo, Seokwon Moon, Yeonhong Park, and Jae W. Lee. 2025. FACIL: Flexible DRAM Address Mapping for SoC-PIM Cooperative On-device LLM Inference. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Press, Piscataway, NJ, USA, 1720–1733. doi:10.1109/HPCA61900.2025.00127

[38] TinyChatEngine. 2023. TinyChatEngine. https://github.com/mit-han-lab/TinyChatEngine.

[39] Jasmina Vasiljevic and Davor Capalija. 2024. Blackhole & TT-Metalium: The Standalone AI Computer and its Programming Model . In *2024 IEEE Hot Chips 36 Symposium (HCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–30. doi:10.1109/HCS61935.2024.10664810

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[41] Zhongkai Yu, Shengwen Liang, Tianyun Ma, Yunke Cai, Ziyuan Nan, Di Huang, Xinkai Song, Yifan Hao, Jie Zhang, Tian Zhi, Yongwei Zhao, Zidong Du, Xing Hu, Qi Guo, and Tianshi Chen. 2024. Cambricon-LLM: A Chiplet-Based Hybrid Architecture for On-Device Inference of 70B LLM. In *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture* (Austin, TX, USA) *(MICRO '24)*. IEEE Press, Piscataway, NJ, USA, 1474–1488. doi:10.1109/MICRO61859.2024.00108