FACIL: Flexible DRAM Address Mapping for SoC-PIM Cooperative On-device LLM Inference

Seong Hoon Seo† Junghoon Kim† Donghyun Lee† Seonah Yoo† Seokwon Moon‡
Yeonhong Park† Jae W. Lee†
†Seoul National University ‡Hanyang University
†{andyseo247, jh0615, eudh1206, ysa9171, ilil96, jaewlee}@snu.ac.kr ‡artichoke42@hanyang.ac.kr

***Abstract*—The rise of on-device inference of large language models (LLMs) is rapidly escalating the demand for memory-intensive operations on edge devices. While DRAM-based processing-in-memory (PIM) is a promising solution for overcoming the memory wall, edge devices require PIM to function both as a compute unit and a memory device due to their limited memory capacity. Such *PIM-enabled memory* complicates the partition and placement of a tensor into DRAM banks in a PIM-operable manner. Notably, we highlight that LLM weights need to be accessible by both PIM and system-on-chip (SoC) processors, as the same weights are used for both SoC-favorable GEMM and PIM-favorable GEMV operations. This necessitates different memory mappings for PIM and SoC processors, leading to potential re-layout costs when switching between the two. To address this challenge, we propose FACIL, a flexible DRAM address mapping solution that efficiently places tensors in DRAM for PIM operations while allowing SoC processors to access the same data using contiguous virtual addresses. FACIL consists of (i) a memory controller that assigns different DRAM address mapping to the page offset bits of each huge page and (ii) a user-level library that determines the appropriate DRAM address mapping. We demonstrate that enabling *re-layout-free* access of both PIM and SoC processor benefits LLM inference on various on-device LLM tasks, including short conversation and code autocompletion, reducing the time-to-first-token by 2.37× and 2.63×, respectively, over the SoC-PIM baseline.**

I. Introduction

The demand for on-device large language models (LLMs) is rapidly growing because they improve user experience by processing data locally, allowing instant responses in real-time applications such as voice assistants and predictive text, even without a stable internet connection. This local processing also enhances privacy and security, as sensitive data remains on the device, mitigating potential risks associated with data transmission to external servers. The inclusion of small LLM inference (GPT-J 6B [82]) in the recent version of the MLPerf Inference benchmark [68] and the release of on-device generative AI tools by major IT vendors [3], [72] are just a few of the many examples that underscore this trend.

On-device LLM inference, which typically processes a single query from a single user at a time, is primarily composed of general matrix-vector multiplications (GEMV) that are highly memory-bound. Processing-in-memory (PIM) is a promising solution to overcome the memory bandwidth wall. Among the various PIM technologies, those that leverage *DRAM bank-level* parallelism by placing a compute unit *near-bank* are emerging as promising candidates for integration into real-world commodity platforms. Notably, all recently introduced production-grade PIM devices adhere to this paradigm [15], [39], [40], [42], [46].

However, adopting near-bank PIM for on-device LLM inference raises a critical question: how can we integrate PIM within the existing memory system? In order to maximize the computational throughput of PIM, weight matrices need to be stored using a specialized data mapping scheme that differs from the conventional ones used by existing memory systems. While supporting such a specialized mapping is relatively straightforward when using a PIM processor as a stand-alone accelerator with a dedicated memory system [28], [41], [74], this approach is impractical for resource-constrained, SoC-based portable devices like mobile phones and laptops. For these platforms, the PIM processor must be integrated into the existing memory system, handling memory requests from other concurrent tasks.

Two specific challenges need to be addressed to realize such integration. First, we must be able to store weight matrices using a PIM-optimized data mapping scheme, which differs from the conventional mapping used by existing memory controllers. Second, SoC processors (CPUs, GPUs, and NPUs) must be able to access the data stored in a PIM-optimized mapping. Such data sharing is crucial because LLM inference involves not only GEMV operations but also GEMM operations, for which PIM processing is not efficient due to their compute-intensive characteristics (i.e., high arithmetic intensity). Ideally, this should be accomplished while maintaining a consistent virtual memory view to leverage existing highly optimized GEMM kernels without extensive modifications. While the first challenge can be addressed through specialized software that allocates memory using huge pages and adjusts the data layout within a page to achieve a PIM-optimized mapping, the second challenge remains unresolved.

To address this unresolved issue, we propose FACIL, a comprehensive solution that provides **f**lexible DRAM **a**ddress mapping for SoC-PIM **c**ooperative **i**nference of **LLM** in a programmer-transparent manner. FACIL holistically augments the memory system, including the paging mechanism in the operating system and the memory controller, to support both conventional and multiple PIM-optimized mapping schemes natively. When allocating memory for data used in PIM computations, FACIL automatically selects the optimal mapping scheme and stores the data accordingly. With FACIL, SoC

processors can access data stored in a PIM-optimized mapping without any changes to the application, as it seamlessly translates the addresses according to the appropriate mapping scheme. With minimal modifications to the operating system and hardware, FACIL significantly improves the responsiveness of on-device LLM inference, which is measured by time-to-first-token (TTFT), compared to the baseline system. This improvement is achieved by eliminating the need to re-layout weight matrices for each GEMM operation executed on SoC processors. According to our evaluation of various platforms, FACIL reduces TTFT by 2.37× on a conversation dataset and by 2.63× on a code autocompletion dataset.

The main contributions of our work are as follows:

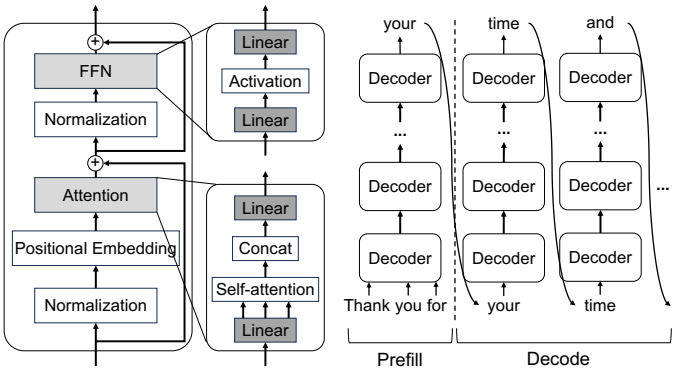
- We identify a critical challenge in achieving efficient data sharing between PIM and SoC processors, which is essential for enhancing the responsiveness of on-device LLM inference using PIM.
- We propose FACIL, a solution that enables efficient, programmer-transparent data sharing between PIM and SoC processors by flexibly selecting the optimal address mapping scheme for PIM computation at huge page granularity.
- We design a non-intrusive augmentation to the paging mechanism and memory controller architecture to natively support multiple address mapping schemes.
- We demonstrate the effectiveness of FACIL across four different platforms on real-world datasets, showcasing significant improvements in responsiveness and inference latency.

II. Background

A. On-device LLM Inference

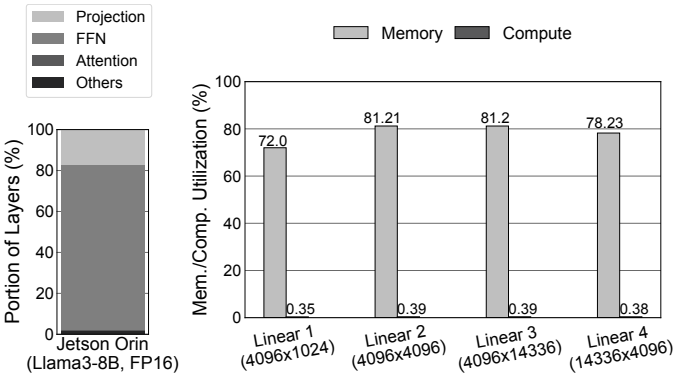
Case for On-device LLM. Performing LLM inference *on-device* offers significant advantages over relying on cloud servers [16], [44], [51], [69]. First, it enables low-latency responses by eliminating the need for data transfer between the device and the server through a network. This results in quicker responses and a better user experience. Second, on-device inference ensures enhanced data privacy and security, as sensitive and confidential user data remains on the device, reducing the risk of data breaches or leaks. Additional benefits include improved personalization and the ability to function offline. The importance of on-device LLM is expected to grow, considering its diverse applications. Major IT vendors have recently introduced LLM services that run on personal portable devices such as mobile phones and laptops. Representative examples include Samsung Galaxy AI [72], Apple Intelligence [3], and Android AI Core [2].

LLM Architecture. Modern LLMs use a common architecture consisting of multiple layers of Transformer [81] decoder blocks, as illustrated in Figure 1(a). Each Transformer decoder block includes several linear operations, attention mechanisms, and additional components such as normalization and positional embeddings. Among these, the linear operations comprise the majority of the model parameters and are typically the most demanding in terms of both computation and memory.



(a) Linear layers within LLM (b) LLM inference process

Fig. 1: LLM architecture and inference



(a) LLM decode time breakdown (b) Memory and compute utilization of linear operations

Fig. 2: Profiling results of decode phase in LLM inference on Jetson AGX Orin using Llama3-8B

LLM Inference. Figure 1(b) illustrates the process of LLM inference. LLM model inference is composed of two phases: prefill (summarization) and decode (generation) phase. For each query, the process starts with the prefill phase, where multiple tokens from the input sequence (e.g., 'thank', 'you', and 'for' in the figure) are processed simultaneously to generate a single output token (e.g., 'your' in the figure). After the prefill phase, this output token is fed back into the model as input, leading to the generation of the next output token. This iterative process continues until the entire sequence is completed, known as the decode phase. Due to the auto-regressive nature of the decode phase, which processes only one token at a time, the linear operations involved in this phase are GEMV, as opposed to being GEMM in the prefill phase. While the prefill phase involves only a single iteration for each query, the decode phase must be iterated tens, if not hundreds or thousands, of times, corresponding to the output sequence length for each query. Therefore, the decode phase often becomes the bottleneck.

LLM Workload Profiling. Figure 2(a) shows the execution time breakdown of the decode phase of generating 64 tokens on the NVIDIA Jetson AGX Orin 64GB with Llama3-8B

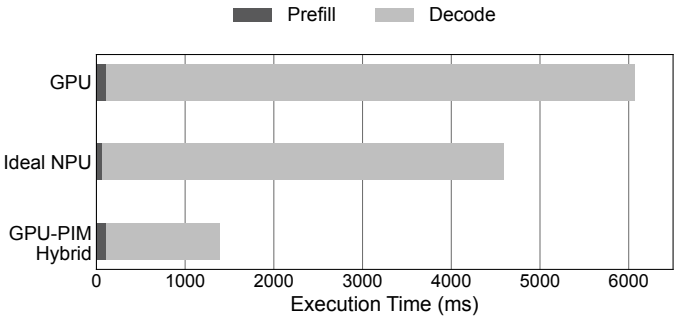


Fig. 3: Potential LLM inference speedup with GEMV operations offloaded to AiM-style PIM processor on Jetson AGX Orin using Llama3-8B

model. As anticipated, more than 90% of the execution time is dedicated to performing linear operations, specifically GEMV, which are highly memory-bound. Figure 2(b) illustrates the compute and memory bandwidth utilization when executing GEMV across the four dimensions employed in the Llama3-8B model. The compute utilization remains very low, below 1%, while the memory bandwidth is heavily utilized.

B. PIM for LLM Inference

Processing-in-memory (PIM), an idea of bringing compute closer to memory, is a powerful solution for accelerating memory-bound operations like GEMV. Among the various classes of DRAM PIM technology based on the location of the processing unit [18], [56], which include in-memory [17], [75], [76], [84], near-bank [21], [27], [40], [45], [47], near-DIMM [7], [36], [37], [43], [66], and buffer die [1], [19], this paper focuses on near-bank DRAM PIMs [21], [27], [40], [45], [47]. This class is noteworthy as it has demonstrated practical feasibility through real-world taped-out prototypes [40], [42], [46] and even commercial products [15].

Figure 3 demonstrates the potential speedup achievable by offloading GEMV operations during the decode phase to AiM-style PIM processor on a Jetson AGX Orin 64GB, using the Llama3-8B model. The details of the PIM configuration are described in Section VI-A. The scenario assumes both the input and output sequence lengths to be 64. By leveraging the significantly higher internal bandwidth provided by PIM, the memory bandwidth bottlenecks associated with the extensive number of GEMV operations are substantially alleviated, leading to a significant end-to-end speedup. We underscore the benefit of integrating PIM by comparing it with a hypothetical, ideal NPU with infinite FLOPS and 100% utilization of the peak memory bandwidth. Despite such optimal assumptions, PIM still achieves 3.32× speedup over the ideal NPU, whose speedup is bounded by the peak memory bandwidth.

C. Data Mapping Problem on PIM Architecture

To fully exploit the internal bank-level parallelism, thereby maximizing the throughput of PIM, a specialized, non-conventional data mapping strategy is essential. Figure 4 illustrates how a matrix should be placed on DRAM for PIM

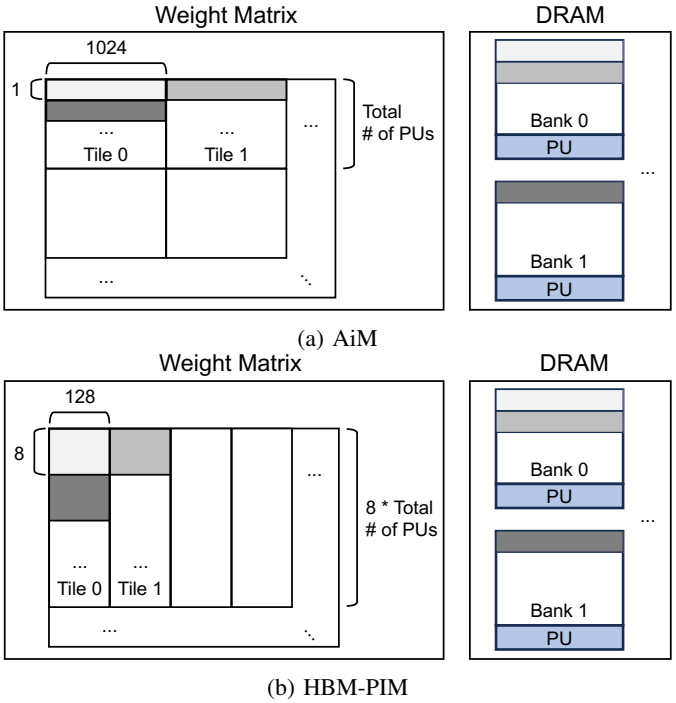


Fig. 4: Illustration of matrix placement on near-bank PIM

to efficiently perform GEMV. How a matrix is mapped to DRAM banks can be expressed using the notion of *chunks* and *tiles*, terms similarly used by prior works [27], [74]. A chunk is the basic unit of computation performed by a single PIM processing unit (PU). The dimension of a chunk is statically determined by the PU architecture, specifically by the tuple of (output register size, input register size). For example, given a data precision of FP/BF16, SK Hynix's Accelerator-in-Memory (AiM) [46] has a chunk dimension of (1, 1024). This is because an input register holds a subset the size of a DRAM row (e.g., 2KB for GDDR6) of the input vector, while an output register holds one element of the output vector. Instead, Samsung's HBM-PIM [42], [73] has a chunk dimension of (8, 128)[1], since it has two sets of 8 general registers, one for input and the other for output, where the size of a register equals the DRAM transfer size (e.g., 32B for HBM2).

Tile is a collection of chunks that are processed by all banks of all channels at the same time. The tile is usually composed as a row-wise concatenation of chunks, as the banks within a rank operate in a lock-step, *all-bank* manner, processing the same command (i.e., accessing the same location within each bank). Aligning the chunks row-wise ensures that the PUs access the same column index of the weight matrix, easing the broadcast [73] or sharing [27] of the input vector.

The optimal placement of a weight matrix for GEMV operation can be summarized as follows. First, the elements within a chunk must be placed contiguously (i.e., at the same DRAM

[1]A set of 8 general registers can, in theory, hold 128 elements of an output vector. However, in practice, each register holds 16 partial sums corresponding to a single element of an output vector due to the lack of a reduction unit [29], [73]. Thus, the chunk dimension is (8, 128), as opposed to being (128, 128).

row) to minimize the row buffer conflict. Second, the banks of a single channel must store different matrix rows in unison, enabling lock-step, all-bank operation of PIM. Finally, it is favorable for a tile to be composed of row-wise concatenation of chunks, assigning each row of a matrix in its entirety to a single bank. This prevents the overhead of reducing the partial sums scattered across banks. One can easily see that the PIM-optimized mapping differs from conventional DRAM address mapping in that it requires a certain amount of data to be contiguously placed within the same bank, and requires data alignment within the banks that act in unison.

III. Goal: Resource-Efficient PIM Integration for On-device LLM Acceleration

The integration of PIM into portable devices like mobile phones and laptops must be implemented in a resource-efficient manner. These devices typically utilize a system-on-chip (SoC) design, and it is essential to integrate PIM directly into the existing SoC platform rather than as a separate accelerator connected via an external interface. Having a separate memory module dedicated solely to PIM processing is impractical for these devices. Assuming this resource-efficient approach, several challenges arise in supporting a PIM-optimized data mapping within the existing memory system. We are constrained to utilize the existing memory controller, which only supports conventional data mapping schemes, even for storing and accessing data for PIM computation. Specifically, the two key challenges are: 1) How can we store weight matrices using a PIM-optimized data mapping scheme? 2) How can we enable efficient access to these weight matrices, stored in a PIM-optimized manner, by both PIM and SoC processors (CPU, GPU, NPU)?

Storing Data with PIM-optimized Mapping Scheme. The first challenge, which is fundamental for making PIM itself feasible, can be addressed by using huge pages [29], [61], [62]. This solution makes a reasonable assumption that the address mapping scheme of the memory controller is made available to those who write software responsible for arranging weight matrices for PIM. Assuming a 2MB huge page, all the necessary bits for interleaving (bank, channel, rank bits) are usually placed within the 21 least significant bits (LSBs), which correspond to the page offset. This is because most DRAM address mapping schemes typically place row bits to the most significant bits (MSBs) [67]. This is particularly true for edge devices like mobile phones and laptops, whose memory systems usually have a limited number of channels and ranks. In such cases, software that tweaks data ordering at the cache line size granularity within each page can store data following any arbitrary interleaving scheme, including PIM-optimized ones.

Data Sharing between PIM and SoC processors. The second challenge, which is rather specific for the case of accelerating LLM with PIM, meanwhile, has not yet been adequately addressed. While PIM processors excel at accelerating GEMV operations, they are not well-suited for GEMM operations, which have much higher arithmetic intensity. For optimal execution of LLM inference, not only the PIM processor but also the SoC processors (CPU, GPU, NPU), which have greater compute capabilities, need to access weight matrices to perform GEMM operations during the prefill phase. This necessitates that SoC processors can access data stored in a PIM-optimized format, which is not supported by the existing memory systems. In particular, widely used BLAS libraries such as Intel oneMKL [30] and NVIDIA cuBLAS [57] view a matrix in virtual address space as a one dimensional array stored in either row-major or column-major order. Breaking this abstraction hinders the usage of highly optimized implementation of kernels provided by the libraries, costing either the remapping of a matrix back into row-/column-major order or implementing handcrafted kernels customized for the layout used.

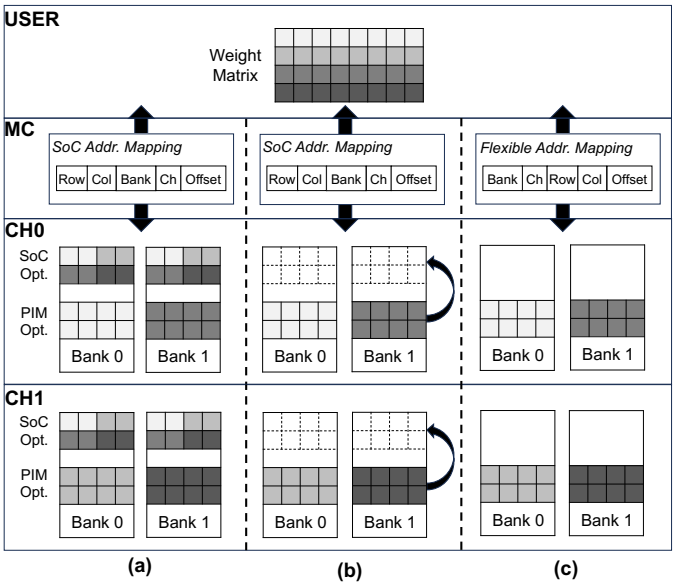


Fig. 5: Three approaches for SoC-PIM cooperative LLM inference: (a) weight duplication, (b) on-demand re-layout and (c) our proposal

There are two possible ways to work around this issue. One approach is to maintain two copies of the weight matrices: one in a conventional mapping scheme and the other in a PIM-optimized mapping scheme (Figure 5(a)). This method takes advantage of the performance benefits of both mappings. However, it is impractical for resource-constrained SoC platforms to incur a $2\times$ increase in memory usage.

A more practical approach is to keep the weight matrices in a PIM-optimized format during the decode phase and, for the prefill phase, to re-layout the matrices to a conventional mapping on demand[2], performing GEMM operations sequentially (Figure 5(b)). This approach would only slightly increase peak memory usage, comparable to the size of the single largest weight matrix. However, the overhead of re-layout slows down

[2]An alternative to such *on-demand* approach is to re-layout all weight matrices at the end of each phase. However, this *all-at-once* approach incurs additional re-layout overhead when transitioning from prefill to decode phase.

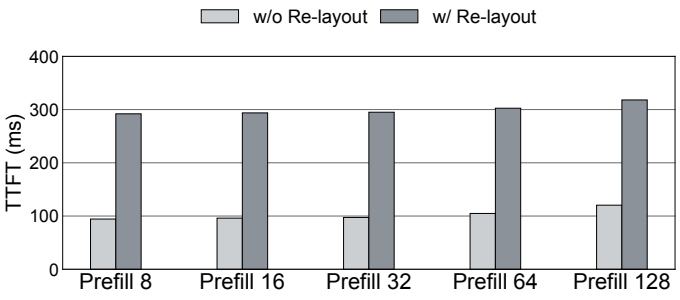


Fig. 6: Increase in TTFT due to re-layout costs with varying input sequence lengths on Jetson AGX Orin using Llama3-8B

the prefill phase, leading to an increased time-to-first-token (TTFT). TTFT is a critical metric that significantly impacts user experience in interactive applications like chatbots, voice assistants, and autocompletion. In such applications, TTFT significantly impacts user-perceived latency more than time-to-last-token (TTLT) [22], [26], the overall inference time. This is because successive output tokens are generated while the user is reading or listening to the previous output. Figure 6 shows the increase in TTFT due to re-layout overhead on the Jetson AGX Orin using the Llama3-8B model with varying input sequence lengths. The TTFT increases by approximately three times, from about 100 ms to 300 ms. This increase is critical because users perceive a system as reacting instantaneously only when the response time is shorter than 100 ms [9]. In fact, to achieve a human-like response time for a voice assistant, which is a popular application of LLMs, it is claimed that the TTFT should be at most around 250 ms [63], [79].

Thus, in this paper, we propose a method that enables SoC processors to efficiently access data stored using a PIM-optimized mapping scheme with minimal hardware and software modifications. By providing user-transparent data sharing between PIM processors and SoC processors, our proposal does not incur an increase in memory footprint or cause a slowdown due to re-layout of tensors. Figure 5(c) illustrates the concept of our proposal. Section IV describes the mechanism of our approach, and Section V provides implementation details.

IV. FACIL: Flexible Address Mapping for SoC-PIM Cooperative Inference of LLM

A. Overview

To provide *re-layout-free* data sharing between the PIM processor and SoC processors, FACIL augments the memory system to support multiple PA-to-DA (Physical Address-to-DRAM Address) mapping schemes, including both conventional and PIM-optimized ones. Specifically, FACIL introduces a specialized memory allocation mechanism called `pimalloc`, which allocates memory regions following a PIM-optimized mapping scheme. Additionally, FACIL allows SoC processors to access these memory regions using only virtual addresses, in a manner that is software-transparent.

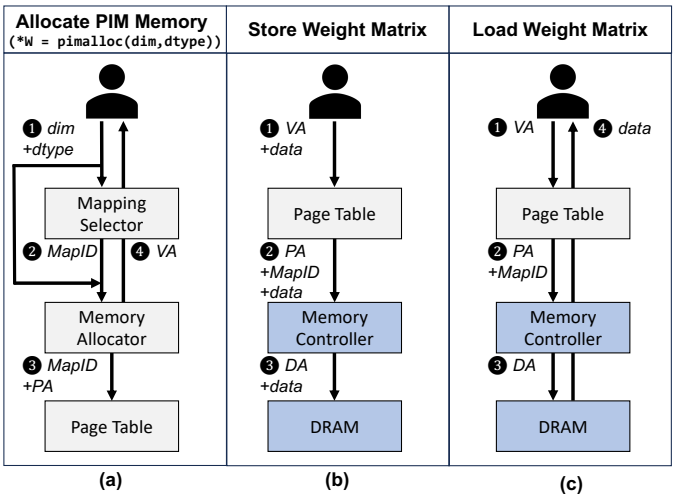


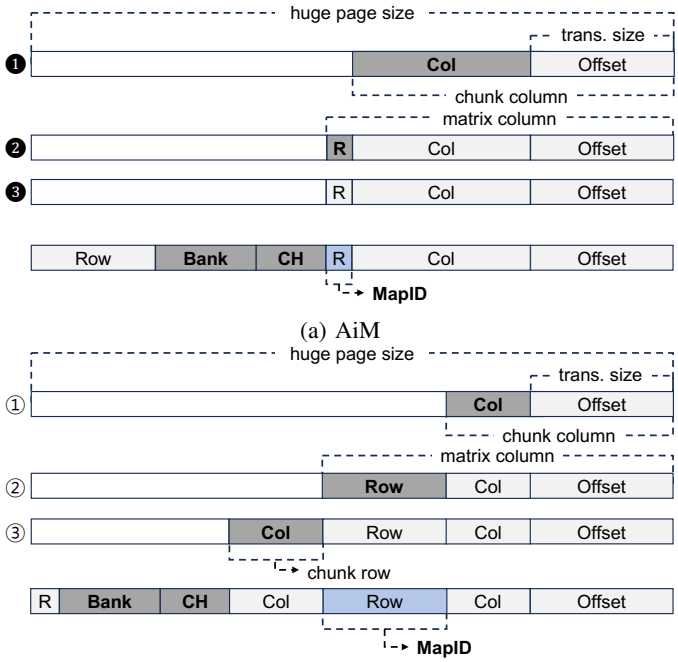
Fig. 7: High level overview of FACIL

Memory Allocation with `pimalloc`. Figure 7(a) illustrates the operation of `pimalloc`, a specialized memory allocation mechanism designed to support PIM-optimized mapping schemes. ❶ Using this interface, a user provides the dimensions and datatype of the weight matrix to a mapping selector. The mapping selector is a user-level software that determines the optimal PA-to-DA mapping based on the matrix configuration, as well as the memory and PIM configuration. ❷ The matrix configuration and the selected MapID are then passed to the OS memory allocator, ❸ which allocates huge pages and records both the physical addresses and the MapID of the allocated pages in the page table. ❹ After these steps, the virtual address is returned to the user.

Programmer-transparent Access to pimalloced Region. Figure 7(b) and (c) illustrate how SoC processors store and read data to and from memory regions allocated by `pimalloc`, respectively. From the programmer's perspective, accessing a `pimalloc` memory region is identical to accessing a normally allocated memory region. For storing data, programmers simply pass the virtual memory address and data to the memory system. For loading data, they only need to pass the virtual memory address. The difference lies between the page table entry and the memory controller. In FACIL, the page table entry contains both physical addresses and MapIDs. Both pieces of information are passed to the memory controller (also along with the data in the case of store). The memory controller, which is augmented with an enhanced address translation module supporting both conventional and PIM-optimized PA-to-DA mapping schemes, performs PA-to-DA mapping according to the given MapID. The data is then stored to or loaded from memory following the selected mapping scheme.

B. Formulation of DRAM Address Mapping

We first explain the generic PA-to-DA mapping that satisfies both PIM-optimized placement and row-major layout, and how it can be represented in the form of *MapID*. Fig. 8 illustrates



(a) AiM

(b) HBM-PIM

Fig. 8: Formulation of the DRAM address mapping for (a) AiM and (b) HBM-PIM

```
int select_mapping(matrix_config,
                   memory_config,
                   pim_config) {

  /* Get matrix, memory, PIM configurations */
  // Matrix
  matrix_col = matrix_config->dim[1]
  dtype = matrix_config->dtype
  row_size = pow(2, ceil(log2(matrix_col))) *
     sizeof(dtype)
  // Memory
  hpage_size = memory_config->hpage_size
  n_ch = memory_config->n_ch
  n_rank = memory_config->n_rank
  n_bank = memory_config->n_bank
  total_bank_count = n_ch * n_rank * n_bank
  // PIM
  chunk_col = pim_config->chunk_col

  /* Determine if partitioning is required */
  memory_per_bank = hpage_size / total_bank_count
  need_partition = memory_per_bank < row_size

  /* Calculate MapID */
  map_id = need_partition ?
     log2(memory_per_bank) : log2(row_size)
  map_id -= log2(chunk_col)

  return map_id
}
```

Fig. 9: Mapping selection algorithm

the following explanation. Note that the figure assumes a single rank memory system (i.e., no DRAM rank bits) for conciseness while FACIL is readily applicable to systems with multiple ranks. Recall from Section II-C, that a chunk must be placed contiguously within a bank for efficient reuse of input and output registers. ❶① Preserving the row-major layout, the PA-to-DA mapping first needs to map a row of a chunk to the same DRAM row. Assuming LPDDR5 DRAM with 32B transfer size and 2KB DRAM row, AiM will have $log_2(1024 * 2/32)$=6 DRAM column bits placed in front of 5 DRAM offset bits. Likewise, in case of HBM-PIM, $log_2(128 * 2/32)$=3 column bits will be placed in front of the DRAM offset bits.

Also recall that it is ideal for a single matrix row to be entirely mapped to a single bank, minimizing the overhead of reducing the partial sums scattered across different banks. ❷② Therefore, it is desirable for a single matrix row to be mapped to a single bank. Therefore, log_2(matrix column / chunk column) DRAM row bits must be placed next.

In case of AiM, whose chunk row dimension is 1, the next matrix row should be mapped to different PUs. ❸ Thus, *PU-changing bits*, which we define as a concatenation of bank, rank, channel bits (i.e., bits that affect the interleaving of banks), should be placed after. ③ On other hand, for HBM-PIM, whose chunk row dimension is 8, 3 column bits should first be prepended, ensuring that the elements within a chunk is placed in the same DRAM row. Only then, will the PU-changing bits be placed. The remaining row bits will fill up the remaining most-significant bits (MSBs) of the page offset.

Given a memory and PIM configuration, which are fixed at the design time of the SoC, the PA-to-DA mapping is determined by the column dimension of the matrix, determining the position of the PU-changing bits. In case of AiM-style PIM, we define the MapID as the number of bits between the PU-changing bits and the chunk column bits, while the MapID for HBM-PIM-style PIM represents the number of bits between the chunk row bits and the chunk column bits.

While there exist numerous possible PA-to-DA mappings applicable to the page offset bits of a huge page, such formulation of mapping limits the number of mappings to the number of positions where the PU-changing bits can be placed between the most-significant bit (MSB) of the page offset and the chunk column bits. When given the DRAM specifications, the theoretical maximum number of MapIDs can be calculated using the following formula:

$$max(MapID) = \log_2 \frac{\text{OS huge page size}}{\text{total bank count} * \text{DRAM transfer size}}$$

Based on this formula, the maximum MapID value for LPDDR5 DRAM in the worst case is 13, assuming a huge page size of 2MB and a DRAM transfer size of 32B. A single channel/rank memory system with 8-bank mode DRAM minimizes the denominator of the formula (i.e., log_2(2MB/(8*32B))=13). The small number of MapIDs allows us to minimize modifications to both the operating system and the memory controller design, as later discussed in Section V.

C. Selection of DRAM Address Mapping

Figure 9 describes how FACIL selects the PA-to-DA mapping upon receiving `pimalloc()` request from a user. While the pseudocode assumes AiM-style PIM, the algorithm is

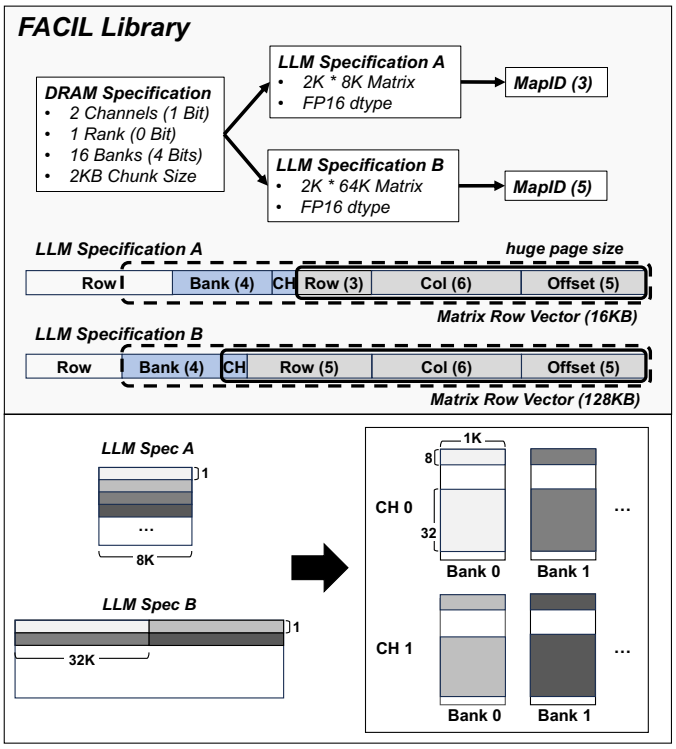


Fig. 10: PA-to-DA mapping for a matrix with large rows

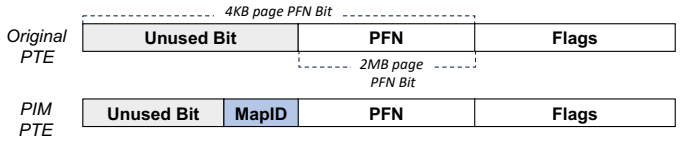


Fig. 11: Modifications to the Page Table Entry

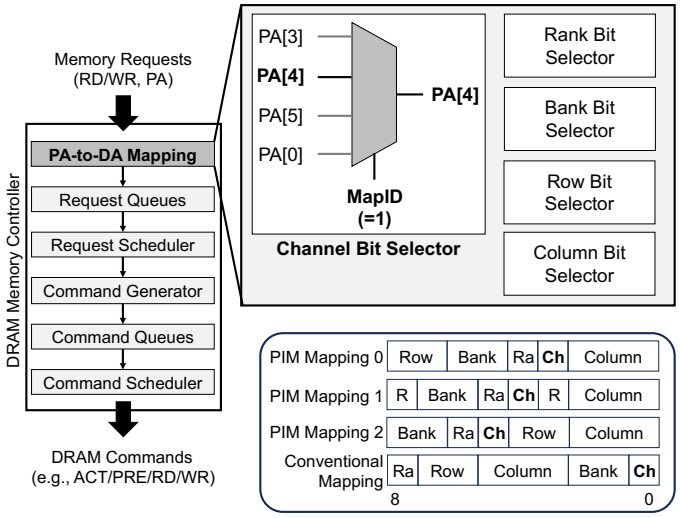


Fig. 12: Memory controller augmentation

also applicable to HBM-PIM-style architectures. First, FACIL retrieves the following three information necessary for determining the MapID (Line 5-18): 1) weight matrix configuration (dimension and data type), 2) memory system configuration (number of channels and ranks), and 3) PIM configuration (chunk dimension). Note that the memory system information can easily be retrieved through interfaces already provided by the operating system such as desktop management interface. Likewise, the chunk dimension can easily be inferred given the high-level architecture of PIM and the type of DRAM.

Then, FACIL checks if an entire matrix row can be mapped to a single bank, or if column-wise partitioning of the matrix is required (Line 20-22). If partitioning is not required, the MapID is determined as the number of bits as described in Section IV-B. On the other hand, if a matrix row is too large to be mapped to a single PU within a single huge page, FACIL places the PU-changing bits to the MSB of the page offset (Line 24-27). Figure 10 shows such an example, where the size of a matrix row vector (e.g., 128KB) is greater than the memory size a huge page can allocate to each PU (e.g., 64KB). FACIL places the PU-changing bits to the MSB of the page offset (i.e., [20:16]). In this case, the matrix row will be mapped to different PUs of a different channel. After PIM operation, the two partial sums of the output vector residing in each channel will be reduced by the SoC processor.

V. Implementation

Implementing FACIL requires modifications to the memory system in addition to the introduction of a mapping selector, which is a user-level method. Specifically, FACIL makes modifications to OS paging mechanism and memory controller. The page table entry should be augmented to store the MapID information for each page and the memory allocator must be able to pass the MapID along with the physical page number to the memory controller. In addition, the memory controller should be able to perform PA-to-DA mapping flexibly based on the given MapID.

A. OS Paging Mechanism Modifications

We extend the `mmap()` system call to optionally receive an additional argument, which is MapID. This MapID is recorded in page table. For recording MapID information, FACIL uses unused bits in the page table entry, avoiding the need for additional memory space. This is feasible because the number of possible PA-to-DA mappings is limited, as discussed in Section IV. Even in the worst case, where 14 additional PA-to-DA mappings must be supported, only four bits are required to represent them. Meanwhile, when using huge pages, there are plenty of unused bits since huge pages require fewer bits for the physical page number than regular pages [64]. For instance, with 4KB regular pages and 2MB huge pages, there are 9 unused bits available ($21 - 12 = 9$), which is sufficient to store the MapID. Figure 11 illustrates this space-efficient partial repurpose of the page table entry. Since a translation lookaside buffer (TLB) entry supports both regular and huge pages, the MapID stored in the unused bits that are available when using huge pages can safely be stored in the TLB entry without requiring any TLB modification.

		Free Memory (relative to model size)			
		2.5×	2.0×	1.5×	1.1×
FMFI	0.0−0.1	10.26s (1.17×)	10.24s (1.16×)	10.24s (1.16×)	10.55s (1.20×)
	0.4−0.5	10.25s (1.16×)	10.23s (1.16×)	11.33s (1.29×)	12.44s (1.41×)
	0.7−0.8	14.48s (1.65×)	15.11s (1.72×)	15.76s (1.79×)	16.72s (1.90×)

TABLE I: Load time of LLM weights when using huge pages under various degrees of memory utilization and fragmentation. The values in parentheses represent the load time normalized to that of baseline that does not use huge pages.

B. Memory Controller Augmentations

FACIL requires only local modifications to the memory controller. Specifically, among the multiple layers of a typical memory controller architecture [34], [54], [70], FACIL necessitates changes to the frontend, which handles the translation of physical addresses to DRAM addresses. In addition to the default PA-to-DA mapping used by the SoC, the memory controller needs to support a small number of additional mappings. This can be implemented with an array of N-to-1 multiplexers, where N is the number of mappings to support. Specifically, five multiplexers are necessary to select the appropriate channel, rank, bank, column, and row bits from the physical address. Figure 12 illustrates this modification, assuming an example where the memory controller needs to support four mappings (three for PIM and one conventional). A simple combinational logic addition to the memory controller, without introducing any memory components, is sufficient to support FACIL.

C. Discussion

Overhead of Huge Page Allocation. We perform an experiment on Jetson AGX Orin 64GB equipped with 1TB of Samsung 980 Pro NVMe SSD to measure the overhead of huge page allocation on the load time of model weights. We select Llama3-8B in FP16 as the target LLM and experiment on various degrees of memory utilization and fragmentation. We represent memory utilization as the free memory size relative to the model size (i.e., 16.2GB), and represent memory fragmentation using the free memory fragmentation index (FMFI) [23]. An FMFI value is in the range of 0 to 1, with a higher value representing a higher degree of memory fragmentation. Table I presents the model load time, where the values in parentheses are normalized to the baseline load time that does not use huge pages. Even in the most unfavorable case, the model load time increases only by 1.90×. Considering that the model load time is a one-time cost that is quickly amortized as multiple rounds of inference are performed on the same model, the overhead of huge page allocation is negligible.

Remaining Challenges. While FACIL aims for the efficient, programmer-transparent data sharing between SoC and PIM processors, challenges remain in fulfilling the integration of PIM into SoC platforms. In particular, the scheduling of PIM and non-PIM memory requests should be carefully handled to minimize the impact on normal processes run on the SoC. We envision that the ideas of prior works that propose efficient memory scheduling methods for CPU-GPU heterogeneous systems with shared memory [8], [85] can be expanded for PIM. An alternative could be the adoption of dual row buffers, as proposed by NeuPIMs [28], such that normal and PIM memory accesses use separate row buffers to prevent DRAM row buffer conflicts.

VI. Evaluation

A. Methodology

Target Platforms and LLMs. We use four SoC-based platforms: NVIDIA Jetson AGX Orin 64GB [60], Apple Macbook Pro [5], Lenovo IdeaPad Slim 5 [48], and Apple iPhone 15 Pro [4]. For each device, we adopt a highly optimized framework to evaluate the execution time of LLMs. Specifically, we use the TinyChatEngine library [51] for Jetson, MLX [25] for MacBook, Intel NPU Acceleration Library [32] for the IdeaPad, and MLX Swift [25] for iPhone. Among the SoC processors, we select the GPU for Jetson, MacBook, and iPhone as well as the NPU for IdeaPad, to serve as the primary processors for executing LLM operations that are not offloaded to PIM processors. These processors are chosen since they demonstrate the best performance for handling LLM tasks on each platform. We use Llama3-8B [53] for Jetson and Macbook, OPT-6.7B [87] for IdeaPad, and Phi-1.5 [50] for iPhone. We use FP16 precision for all cases. Table II summarizes the specifications of the evaluated platforms and models. Note that some of the values in Table II are retrieved from third-party sources [13], [14], [71], [77], [78], due to the absence of such information in the official documentations provided by the manufacturers [6], [31], [33], [58], [59].

PIM Simulation. For each of the device, we assume that the memory is augmented with an AiM-style PIM, where 16 banks in each rank shares an input register (i.e., global buffer) the size of a DRAM row (2KB). We also assume that each channel is composed of two ranks. We use an open source PIM simulator [28], which is based on DRAMsim [49], to measure the performance of PIM. While keeping the core components of the simulator intact, we modified the simulator to support LPDDR5/X memory. We adopt the LPDDR5/X timing parameters from the JEDEC standard [35].

FACIL. We measure the latency of FACIL as the sum of the *conservatively scaled* GEMM time measured on the real device and the GEMV time simulated on the PIM simulator. Since GEMM operations are performed on a PIM-optimized layout in the case of FACIL, rather than using the conventional address mapping, there can be potential side effects on performance. To account for these side effects, we measure the performance impact of the mapping change on GEMM operations using GPGPU-Sim [38] and ONNXim [24], each modeling GPU (Jetson, Macbook, iPhone) and NPU (IdeaPad), respectively. We configure each XPU based on publicly known information, mainly number of cores and peak FLOPS. We evaluate with different weight matrix dimensions and prefill lengths. Table III summarizes the results. We conservatively

Platform	Primary SoC Processor			Memory					LLM		
	Processor Name	Type	Peak Throughput (TFLOPS, FP16)	DRAM Type	Data Rate (Mbps)	Bus Width (bits)	Capacity (GB)	Peak BW (GB/s)	Model Name	Framework/ Library	Precision
NVIDIA Jetson AGX Orin 64GB	Ampere CUDA/Tensor Cores	GPU	42.5	LPDDR5	6400	256	64	204.8	Llama3-8B	TinyChatEngine	FP16
Apple Macbook Pro	M3 Max	GPU	28.4	LPDDR5	6400	512	64	409.6	Llama3-8B	MLX	FP16
Lenovo IdeaPad Slim 5	Intel Core Ultra 7 155H	NPU	5.6	LPDDR5X	7467	64	32	59.7	OPT-6.7B	Intel NPU Library	FP16
Apple iPhone 15 Pro	A17 Pro	GPU	4.29	LPDDR5	6400	64	8	51.2	Phi-1.5	MLX Swift	FP16

TABLE II: Specifications of the evaluated platforms and models

	Jetson Orin (GPU)			Macbook Pro (GPU)		
Prefill Length	4	16	64	4	16	64
Q/O Proj.	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%
K/V Proj.	0.1%	0.3%	0.2%	0.0%	**0.1%**	**0.1%**
FC1	0.9%	1.1%	**2.1%**	0.0%	0.0%	0.0%
FC2	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%
	IdeaPad Slim 5 (NPU)			iPhone 15 Pro (GPU)		
Prefill Length	4	16	64	4	16	64
Q/K/V/O Proj.	0.8%	0.9%	0.9%	0.5%	0.4%	**1.6%**
FC1	0.9%	**1.1%**	0.8%	0.0%	0.0%	0.0%
FC2	0.2%	0.1%	0.3%	0.1%	0.0%	0.0%

TABLE III: Performance slowdown of GEMM on PIM-optimized layout experimented on various prefill lengths

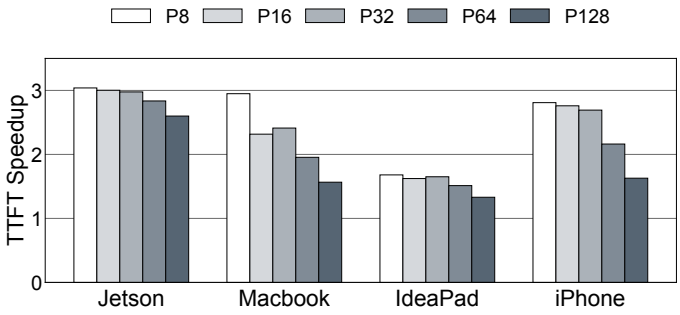


Fig. 13: TTFT speedup of FACIL over the SoC-PIM hybrid baseline with varying prefill length

choose the worst-case slowdown for each device, 2.1%, 0.1%, 1.1%, and 1.6% for Jetson, Macbook, IdeaPad, and iPhone, respectively, and scale its GEMM latency by these ratios for all FACIL performance reports.

Baseline. We define a SoC-PIM hybrid baseline as a PIM-enabled device without the features of FACIL. The baseline holds a single copy of LLM weights, stored in a PIM-optimized layout. The baseline offloads the prefill phase (i.e., GEMM) to the SoC processor and the decode phase (i.e., GEMV) to PIM. Transitioning from the PIM-optimized layout to conventional layout incurs a *re-layout cost*. The latency of the SoC-PIM hybrid baseline is computed as the sum of 1) GEMM time measured on the real device, 2) GEMV time simulated on the PIM simulator, and 3) re-layout time simulated on the DRAM simulator. We estimate the re-layout cost using DRAMSim [49], modeling only the memory access time required to read data stored in one layout and write it into another layout. For the DRAM address mapping of the SoC, we assume a typical mapping of *row:rank:column:bank:channel*, which we verify achieves near-peak sequential read bandwidth. Note that this estimation is conservative in two aspects. First, the re-layout cost only considers the memory copy overhead, excluding the overhead of rearranging data ordering within each page. Second, we assume that full memory bandwidth is available, while in reality, some bandwidth may be utilized by other processes.

B. Evaluation on a Single Query

TTFT. Figure 13 shows the time-to-first-token (TTFT) improvement of a single query on varying prefill lengths, compared to the SoC-PIM hybrid baseline. FACIL achieves a geomean speedup of 2.89×, 2.19×, 1.55×, and 2.36× at Jetson, Macbook, IdeaPad, and iPhone respectively, across different prefill lengths. The TTFT speedup is inversely proportional to the prefill length since the longer a prefill takes, the more the re-layout cost becomes amortized. Meanwhile, the degree of how fast the speedup diminishes as the prefill length increases differs among devices. This difference stems from the arithmetic intensity (i.e., FLOP/byte) of the ridge points of each device in the roofline model [83], the minimum arithmetic intensity required to achieve the peak FLOPS (i.e., Peak FLOPS/Peak Bandwidth). While the arithmetic intensity of GEMM increases as the prefill length increases, its latency sublinearly increases until its arithmetic intensity hits the arithmetic intensity of the ridge point. Thus, the higher the arithmetic intensity of the ridge point is, the slower the TTFT speedup diminishes. The ridge point arithmetic intensity of Macbook (i.e., 69.3) and iPhone (i.e., 83.8) are lower than that of Jetson (i.e., 207.5) and IdeaPad (i.e., 93.8), thus showing faster diminishment of speedup, as depicted in Figure 13.

TTLT. Figure 14 depicts the time-to-last-token (TTLT) improvement of a single query inference across varying combinations of prefill and decode lengths compared to the SoC-PIM hybrid baseline. Unlike TTFT, which depends solely on the prefill phase, TTLT is heavily influenced by the decode length due to its auto-regressive property. While the TTLT speedup of FACIL is largely amortized during long decode phases, FACIL still achieves significant speedup for decode lengths up to 64, with an improvement of approximately 10%.

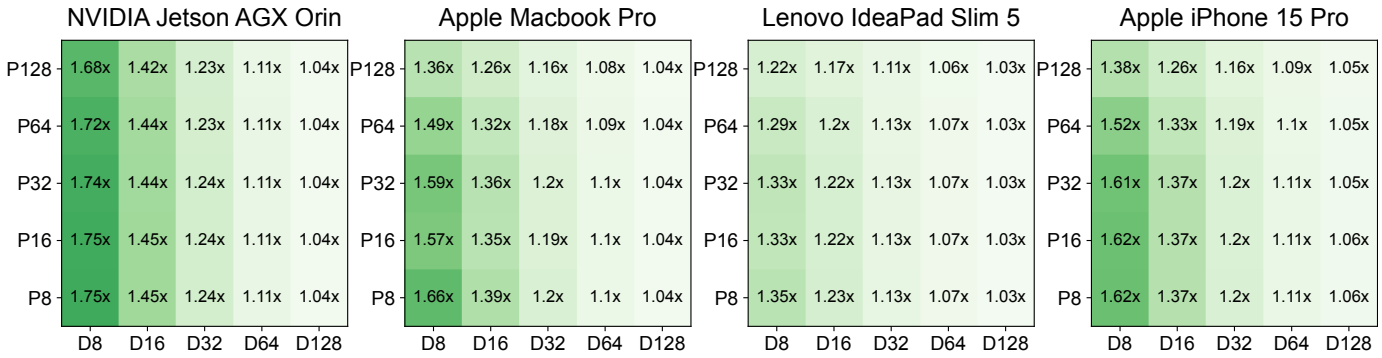


Fig. 14: TTLT speedup of FACIL over the SoC-PIM hybrid baseline with varying prefill-to-decode ratio

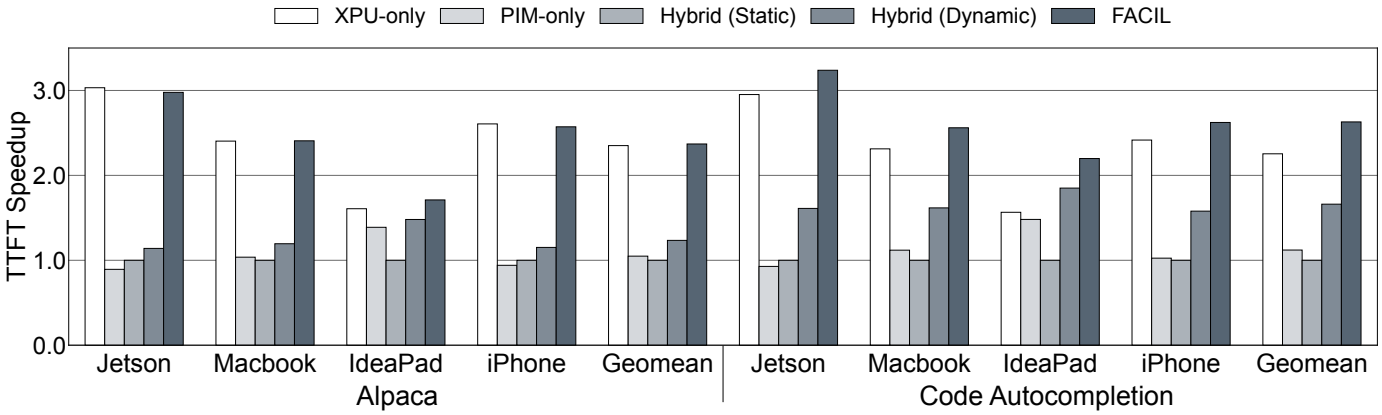


Fig. 15: Normalized TTFT speedup of FACIL on real-world datasets

C. Evaluation on Real-World Datasets

We evaluate our approach on two real-world datasets: Alpaca [80] and RealHumanEval [55]. Alpaca is a dataset composed of real-world inputs paired with output texts generated by GPT-3.5, which is representative of an LLM-based virtual assistant. For code autocompletion, we utilize the "autocompletion" subset of the RealHumanEval dataset, which includes interaction logs between programmers and LLMs with code autocompletion support. Both datasets represent key applications of on-device LLMs where responsiveness is of high significance. We randomly sample 1% and 10% of each dataset, tokenize these samples, and use the resulting number of tokens as input and output lengths for our experiments.

Since real-world datasets have varying prefill lengths, we introduce a new optimization technique that allows the system to offload the GEMM operations of the prefill phase to the optimal hardware (i.e., SoC vs. PIM) based on the prefill length. For example, when the prefill lengths are too small, resulting in tall-and-skinny GEMM operations that are better executed on PIM processors, we choose PIM processors for execution. We profile the prefill execution time of SoC and PIM beforehand to determine the threshold at which SoC becomes faster than PIM. We henceforth denote this highly optimized baseline as *hybrid dynamic*, and term the original baseline, which performs every GEMM on SoC, as *hybrid static*. Note that FACIL in Figure 15 and Figure 16 refers to the version with this optimization applied.

The result shows that our scheme achieves a geomean TTFT speedup of 2.37× and 2.63× for Alpaca and code autocompletion dataset respectively, and a TTLT speedup of 1.20× for both datasets over the static baseline. While software optimization itself offers non-negligible speedup compared to the static baseline, our scheme still outperforms the dynamic baseline by a large margin. We achieve slightly better TTFT speedup compared to SoC-only inference because dataset contains queries with small prefill length, in which PIM outperforms the SoC processor. Note that while SoC-only inference can provide fast TTFT, it greatly suffers in TTLT because of its relative slowdown during the memory-bound decode phase. In detail, FACIL achieves 3.55× and 3.58× TTLT speedup compared to the SoC-only inference on Alpaca and code autocompletion dataset, respectively.

FACIL performs best on Jetson among the four platforms, while showing the least speedup on the IdeaPad. This can be explained by memory bandwidth utilization of each platform. We measured the memory bandwidth utilization of the GEMV kernel on each platform by dividing the total memory access by the GEMV execution time. Jetson, MacBook, and iPhone all demonstrated high memory utilization (i.e. 76.3%, 88.3%,

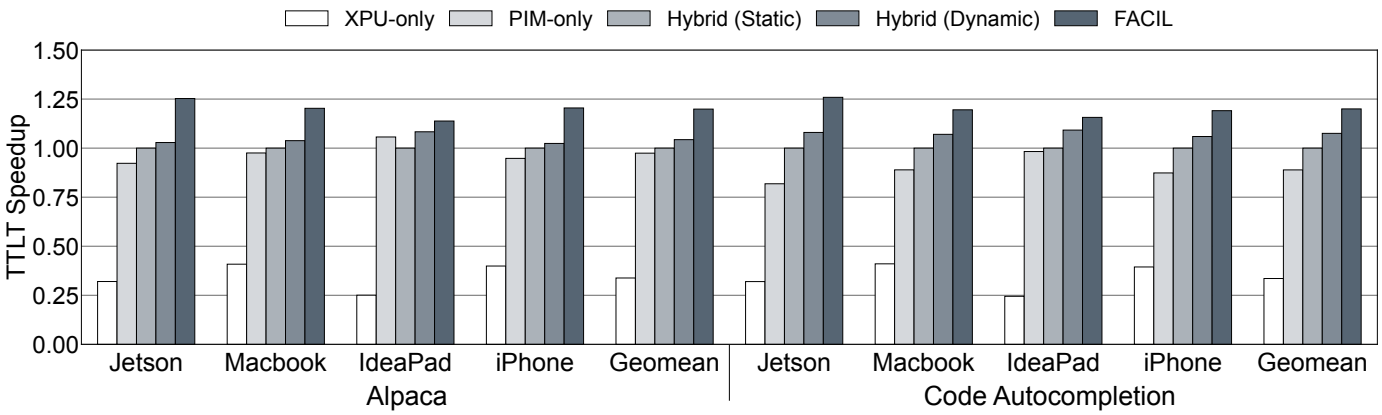


Fig. 16: Normalized TTLT speedup of FACIL on real-world datasets

74.6%), each surpassing 70% of their maximum memory bandwidth. In contrast, the IdeaPad exhibited a notably low memory bandwidth utilization of 33.3%. This relatively low capability of the IdeaPad in utilizing memory bandwidth slows down the prefill phase with skinny GEMM, which is mostly memory-bound. Consequently, the portion of memory re-layout cost in the total execution time decreases, leading to reduced performance of FACIL in both TTFT and TTLT.

VII. Related Work

PIM for LLM. Since the advent of taped-out near-bank DRAM PIMs [15], [21], [42], [46], demonstrating its potential of accelerating memory-bound kernels mainly composed of GEMV operations, numerous works have proposed PIM-augmented accelerators to improve the performance of LLM inference [12], [28], [40], [41], [65], [74]. While these works target server-scale LLM inference and typically design a discrete accelerator where PIM is tightly coupled from the design stage, we instead focus on the on-device LLM inference for edge devices, tackling the challenge of efficient data sharing between the existent SoC processor and PIM. We highlight the difference of FACIL with IANUS [74] and NeuPIMs [28] that stems from the difference in the target platform (i.e., server vs. edge). First, while IANUS proposes a global DRAM address mapping scheme for the NPU-PIM accelerator, FACIL leaves the mapping for non-PIM data intact, only modifying the mapping for the SoC-PIM shared data. Second, while NeuPIMs focuses on the parallel execution of GEMM and GEMV operations to serve batched inference, FACIL targets single-batch inference, where GEMM and GEMV are performed in discrete phases. For SoC-based platforms, the major challenge of PIM lies in preserving programmer transparency while achieving the PIM-optimized data placement.

Integration of PIM. There are two main ways of integrating PIM into an existing device. The first is to reserve a certain number of ranks or banks exclusively for PIM, an approach adopted by MI100-PIM [40] and Chopim [11]. While this approach saves the burden of handling the interleaving of DRAM accesses issued by the SoC and PIM processors, allocating distinct memory space for SoC and PIM in edge platforms with limited resources is costly. The alternative is to integrate PIM into the main memory system, the category Stepstone PIM [10] falls into. The main distinction of FACIL from Stepstone PIM is that the former locally modifies the DRAM address mapping for the SoC-PIM shared matrices, while the latter preserves the original DRAM address mapping. FACIL trades in minimal modifications for optimal data placement for PIM-offloaded GEMV operations, minimizing the cost of inter-bank reduction.

Modification of DRAM Address Mapping. Several works have proposed modification to the DRAM address mapping [20], [52], [86] to better utilize the parallelism provided by the internal DRAM structure. DReAM [20] highlights that different workloads exhibit varying memory access patterns, necessitating distinct optimal address mappings. To identify the best address mapping for each pattern, the paper suggested real-time analysis of workload memory access patterns. It also proposed working with the memory controller to regularly update the address mapping table, leading to enhanced performance. Liu et al. [52] developed a method to profile memory address patterns after noticing distinct differences between CPU and GPU access patterns. Their experimental results demonstrated that adopting a mapping scheme optimized for GPUs enhances both energy efficiency and overall performance. Software-defined address mapping (SDAM) [86] is a technique that shares the high-level idea of enabling a user-level program to modify the PA-to-DA mapping at certain granularity. SDAM modifies DRAM address mapping to enhance channel parallelism in 3D memory. While it offers a wide range of PA-to-DA mapping options, it necessitates sophisticated additional hardware and significant operating system modifications.

VIII. Conclusion

FACIL presents a solution to address the challenges of SoC-PIM cooperative on-device LLM inference, particularly for edge devices with limited memory capacity. By enabling flexible DRAM address mapping, FACIL efficiently manages the complex requirements of both PIM and SoC processors, allowing for optimized data sharing and improved performance

in memory-intensive operations. The proposed system, comprising a specialized memory controller and user-level library, demonstrates significant advancements in responsiveness and inference latency across various real-world datasets and platforms. As the demand for on-device LLM inference continues to grow, driven by the need for enhanced user experience, privacy, and offline functionality, FACIL's approach to seamlessly integrating PIM technology with existing memory systems could play a crucial role in future AI applications.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grants funded by the Korea government (MSIT) (RS-2024-00340008, RS-2024-00405857) and by the MSIT (Ministry of Science, ICT), Korea, under the Global Scholars Invitation Program (RS-2024-00456287) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation). Jae W. Lee is the corresponding author.

REFERENCES

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[2] Android Developers, "Google AI Edge SDK for Gemini Nano," https://developer.android.com/ai/aicore.

[3] Apple, "Apple Intelligence," https://www.apple.com/apple-intelligence/.

[4] Apple, "iPhone 15 Pro," https://www.apple.com/iphone-15-pro/.

[5] Apple, "Macbook Pro," https://www.apple.com/macbook-pro/.

[6] Apple, "MacBook Pro (16-inch, Nov 2023) - Technical Specifications," https://support.apple.com/en-us/117737.

[7] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[8] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[9] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "THE INFORMATION VISUALIZER, AN INFORMATION WORKSPACE," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 1991.

[10] B. Y. Cho, J. Jung, and M. Erez, "Accelerating Bandwidth-Bound Deep Learning Inference with Main-Memory Accelerators," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[11] B. Y. Cho, Y. Kwon, S. Lym, and M. Erez, "Near Data Acceleration with Concurrent Host Access," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[12] J. Choi, J. Park, K. Kyung, N. S. Kim, and J. H. Ahn, "Unleashing the Potential of PIM: Accelerating Large Batched Inference of Transformer-Based Generative Models," *IEEE Computer Architecture Letters*, 2023.

[13] CPU-Monkey, "Apple A17 Pro (6 GPU Cores)," https://www.cpu-monkey.com/en/igpu-apple_a17_pro_6_gpu_cores.

[14] CPU-Monkey, "Apple M3 Max (40 Core)," https://www.cpu-monkey.com/en/igpu-apple_m3_max_40_core.

[15] F. Devaux, "The true Processing In Memory accelerator," in *IEEE Hot Chips 31 Symposium (HCS)*, 2019.

[16] S. Dhar, J. Guo, J. J. Liu, S. Tripathi, U. Kurup, and M. Shah, "A Survey of On-Device Machine Learning: An Algorithms and Learning Theory Perspective," *ACM Transactions on Internet of Things*, vol. 2, no. 3, 2021.

[17] D. W. Fei Gao, Georgios Tziantzioulis, "ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[18] D. Fujiki, X. Wang, A. Subramaniyan, and R. Das, *In-/Near-Memory Computing*. Springer, 2021.

[19] M. Gao, G. Ayers, and C. Kozyrakis, "Practical Near-Data Processing for In-memory Analytics Frameworks," in *International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[20] M. Ghasempour, A. Jaleel, J. D. Garside, and M. Luján, "DReAM: Dynamic Re-arrangement of Address Mapping to Improve the Performance of DRAMs," in *Proceedings of the Second International Symposium on Memory Systems*, 2016.

[21] J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System," *IEEE Access*, vol. 10, 2022.

[22] Google Cloud, "Best practices with large language models (LLMs)," https://cloud.google.com/vertex-ai/generative-ai/docs/learn/prompt-best-practices.

[23] M. Gorman and A. Whitcroft, "The what, the why and the where to of anti-fragmentation," in *Ottawa Linux Symposium*, 2006.

[24] H. Ham, W. Yang, Y. Shin, O. Woo, G. Heo, S. Lee, J. Park, and G. Kim, "ONNXim: A Fast, Cycle-level Multi-core NPU Simulator," *arXiv preprint arXiv:2406.08051*, 2024.

[25] A. Hannun, J. Digani, A. Katharopoulos, and R. Collobert, "MLX: Efficient and flexible machine learning on apple silicon," https://github.com/ml-explore, 2023.

[26] G. N. Harel Gal, Eitan Sela and M. Mayer, "Build safe and responsible generative AI applications with guardrails," https://aws.amazon.com/blogs/machine-learning/build-safe-and-responsible-generative-ai-applications-with-guardrails/.

[27] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. N. Vijaykumar, "Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[28] G. Heo, S. Lee, J. Cho, H. Choi, S. Lee, H. Ham, G. Kim, D. Mahajan, and J. Park, "NeuPIMs: NPU-PIM Heterogeneous Acceleration for Batched LLM Inferencing," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[29] M. A. Ibrahim, M. Islam, and S. Aga, "Balanced Data Placement for GEMV Acceleration with Processing-In-Memory," *arXiv preprint arXiv:2403.20297*, 2024.

[30] Intel, "Intel oneAPI Math Kernel Library - Data Parallel C++ Developer Reference," https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-dpcpp/2024-1/matrix-storage.html.

[31] Intel, "Intel® Core™ Ultra 7 Processor 155H," https://www.intel.com/content/www/us/en/products/sku/236847/intel-core-ultra-7-processor-155h-24m-cache-up-to-4-80-ghz/specifications.html.

[32] Intel, "Intel NPU Acceleration Library," https://github.com/intel/intel-npu-acceleration-library, 2024.

[33] Intel, "Intel® Core™ Ultra Processors (PS Series) Datasheet," https://www.intel.com/content/www/us/en/content-details/819636/intel-core-ultra-processors-ps-series-datasheet.html, 2024.

[34] B. Jacob, D. Wang, and S. Ng, *Memory Systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[35] JEDEC committee, "LOW POWER DOUBLE DATA RATE (LPDDR) 5/5X," https://www.jedec.org/standards-documents/docs/jesd209-5c, 2023.

[36] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, "RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[37] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon, K. Kim, J. Jung, I. Yun, S. J. Park, H. Park, J. Song, J. Cho, K. Sohn, N. S. Kim, and H.-H. S. Lee, "Near-Memory Processing in Action: Accelerating Personalized Recommendation With AxDIMM," *IEEE Micro*, vol. 42, no. 1, 2022.

[38] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[39] J. H. Kim, S.-H. Kang, S. Lee, H. Kim, Y. Ro, S. Lee, D. Wang, J. Choi, J. So, Y. Cho, J. Song, J. Cho, K. Sohn, and N. S. Kim,

"Aquabolt-XL HBM2-PIM, LPDDR5-PIM With In-Memory Processing, and AXDIMM With Acceleration Buffer," *IEEE Micro*, vol. 42, no. 3, 2022.

[40] J. H. Kim, Y. Ro, J. So, S. Lee, S.-h. Kang, Y. Cho, H. Kim, B. Kim, K. Kim, S. Park, J.-S. Kim, S. Cha, W.-J. Lee, J. Jung, J.-G. Lee, J. Lee, J. Song, S. Lee, J. Cho, J. Yu, and K. Sohn, "Samsung PIM/PNM for Transformer Based AI : Energy Efficiency on PIM/PNM Cluster," in *IEEE Hot Chips 35 Symposium (HCS)*, 2023.

[41] Y. Kwon, G. Kim, N. Kim, W. Shin, J. Won, H. Joo, H. Choi, B. An, G. Shin, D. Yun, J. Kim, C. Kim, I. Kim, J. Park, C. Park, Y. Song, B. Yang, H. Lee, S. Park, W. Lee, S. Lee, K. Kim, D. Kwon, C. Jeong, J. Kim, E. Lim, and J. Chun, "Memory-Centric Computing with SK Hynix's Domain-Specific Memory," in *IEEE Hot Chips 35 Symposium (HCS)*, 2023.

[42] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim, Y. Cho, J. G. Kim, J. Choi, H.-S. Shin, J. Kim, B. Phuah, H. Kim, M. J. Song, A. Choi, D. Kim, S. Kim, E.-B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. Song, J. Youn, K. Sohn, and N. S. Kim, "25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2021.

[43] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[44] P. Lawlor and J. Chang, "5 benefits of on-device generative AI," https://www.qualcomm.com/news/onq/2023/08/5-benefits-of-on-device-generative-ai, 2023.

[45] J. Lee, J. H. Ahn, and K. Choi, "Buffered Compares: Excavating the Hidden Parallelism Inside DRAM Architectures with Lightweight Logic," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.

[46] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, K. Vladimir, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, J. Lee, D. Ko, Y. Jun, K. Cho, I. Kim, C. Song, C. Jeong, D. Kwon, J. Jang, I. Park, J. Chun, and J. Cho, "A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2022.

[47] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product," in *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[48] Lenovo Product Specifications Reference, "IdeaPad Slim 5 16IMH9," https://psref.lenovo.com/Product/IdeaPad/IdeaPad_Slim_5_16IMH9.

[49] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.

[50] Y. Li, S. Bubeck, R. Eldan, A. D. Giorno, S. Gunasekar, and Y. T. Lee, "Textbooks Are All You Need II: phi-1.5 technical report," https://arxiv.org/abs/2309.05463, 2023.

[51] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, "AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration," in *Proceedings of the Seventh Annual Conference on Machine Learning and Systems (MLSys)*, 2024.

[52] Y. Liu, X. Zhao, M. Jahre, Z. Wang, X. Wang, Y. Luo, and L. Eeckhout, "Get Out of the Valley: Power-Efficient Address Mapping for GPUs," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 166–179.

[53] Meta, "Llama 3," https://llama.meta.com/llama3, 2024.

[54] R. Mirosanlou, D. Guo, M. Hassan, and R. Pellizzoni, "MCsim: An Extensible DRAM Memory Controller Simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 105–109, 2020.

[55] H. Mozannar, V. Chen, M. Alsobay, S. Das, S. Zhao, D. Wei, M. Nagireddy, P. Sattigeri, A. Talwalkar, and D. Sontag, "The RealHumanEval: Evaluating Large Language Models' Abilities to Support Programmers," *arXiv preprint arXiv:2404.02806*, 2024.

[56] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, *A Modern Primer on Processing in Memory*. Springer Nature Singapore, 2023, pp. 171–243.

[57] NVIDIA, "cuBLAS," https://docs.nvidia.com/cuda/cublas/.

[58] NVIDIA, "NVIDIA Jetson AGX Orin Series Data Sheet," https://www.diamondsystems.com/files/binaries/Jetson_AGX_Orin_DS-10662-001_v1.2.pdf.

[59] NVIDIA, "NVIDIA Jetson AGX Orin Series Technical Brief," https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf.

[60] NVIDIA, "NVIDIA Jetson Orin," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/.

[61] G. F. Oliveira, E. G. Esposito, J. Gómez-Luna, and O. Mutlu, "PUMA: Efficient and Low-Cost Memory Allocation and Alignment Support for Processing-Using-Memory Architectures," *arXiv preprint arXiv:2403.04539*, 2024.

[62] G. F. Oliveira, A. Olgun, A. G. Yağlıkçı, F. N. Bostancı, J. Gómez-Luna, S. Ghose, and O. Mutlu, "MIMDRAM: An End-to-End Processing-Using-DRAM System for High-Throughput, Energy-Efficient and Programmer-Transparent Multiple-Instruction Multiple-Data Computing," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024.

[63] OpenAI, "Hello, GPT-4o!" https://openai.com/index/hello-gpt-4o/, 2024.

[64] C. H. Park, S. Cha, B. Kim, Y. Kwon, D. Black-Schaffer, and J. Huh, "Perforated Page: Supporting Fragmented Memory Allocation for Large Pages," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 913–925.

[65] J. Park, J. Choi, K. Kyung, M. J. Kim, Y. Kwon, N. S. Kim, and J. H. Ahn, "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 (ASPLOS)*, 2024.

[66] J. Park, B. Kim, S. Yun, E. Lee, M. Rhu, and J. H. Ahn, "TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory," in *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[67] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *25th USENIX Security Symposium (SEC)*, 2016.

[68] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "MLPerf Inference Benchmark," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[69] V. J. Reddi, "Machine Learning Systems with TinyML," https://harvard-edge.github.io/cs249r_book/, 2024.

[70] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE computer architecture letters*, vol. 10, no. 1, pp. 16–19, 2011.

[71] J. Rossignol, "iPhone 15 Pro and iPhone 15 Pro Max Feature Increased 8GB of RAM," https://www.macrumors.com/2023/09/12/iphone-15-pro-models-have-8gb-ram/, 2023.

[72] Samsung, "Galaxy AI," https://www.samsung.com/us/galaxy-ai/.

[73] Samsung Advanced Institute of Technology, https://github.com/SAITPublic/PIMSimulator.

[74] M. Seo, X. T. Nguyen, S. J. Hwang, Y. Kwon, G. Kim, C. Park, I. Kim, J. Park, J. Kim, W. Shin, J. Won, H. Choi, K. Kim, D. Kwon, C. Jeong, S. Lee, Y. Choi, W. Byun, S. Baek, H.-J. Lee, and J. Kim, "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 (ASPLOS)*, 2024.

[75] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 185–197.

[76] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit:

In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[77] R. Smith, "Apple Announces M3 SoC Family: M3, M3 Pro, and M3 Max Make Their Marks," https://www.anandtech.com/show/21116/apple-announces-m3-soc-family-m3-m3-pro-and-m3-max-make-their-marks.

[78] O. Sohail, "iPhone 15 Pro, iPhone 15 Pro Max Feature The World's First First 'D1B' LPDDR5 DRAM Chips From Micron," https://wccftech.com/iphone-15-pro-max-feature-first-ever-micron-d1%ce%b2-lpddr5-dram-chips/, 2023.

[79] T. Stivers, N. J. Enfield, P. Brown, C. Englert, M. Hayashi, T. Heinemann, G. Hoymann, F. Rossano, J. P. de Ruiter, K.-E. Yoon, and S. C. Levinson, "Universals and cultural variation in turn-taking in conversation," *Proceedings of the National Academy of Sciences*, vol. 106, no. 26, 2009.

[80] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, "Stanford Alpaca: An Instruction-following LLaMA model," https://github.com/tatsu-lab/stanford_alpaca, 2023.

[81] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[82] B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model," https://github.com/kingoflolz/mesh-transformer-jax, 2021.

[83] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures]," *Communications of the ACM*, vol. 52, no. 4, 2009.

[84] X. Xin, Y. Zhang, and J. Yang, "ELP2IM: Efficient and Low Power Bitwise Operation Processing in DRAM," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[85] Y. Xu, M. E. Belviranli, X. Shen, and J. Vetter, "PCCS: Processor-Centric Contention-aware Slowdown Model for Heterogeneous System-on-Chips," in *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[86] J. Zhang, M. Swift, and J. J. Li, "Software-Defined Address Mapping: A Case on 3D Memory," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[87] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "OPT: Open Pre-trained Transformer Language Models," https://arxiv.org/abs/2205.01068, 2022.