NeuPIMs: NPU-PIM Heterogeneous Acceleration for Batched LLM Inferencing

Guseul Heo Sangyeop Lee Jaehong Cho Hyunmin Choi Sanghyeon Lee
Hyungkyu Ham[†] Gwangsun Kim[†] Divya Mahajan[§] Jongse Park

KAIST, [†]POSTECH, [§]Georgia Institute of Technology

{gsheo, sangyeop-lee, jhcho, hmchoi, leesh6796}@casys.kaist.ac.kr
{hhk971, g.kim}@postech.ac.kr divya.mahajan@gatech.edu jspark@casys.kaist.ac.kr

Abstract

Modern transformer-based Large Language Models (LLMs) are constructed with a series of decoder blocks. Each block comprises three key components: (1) QKV generation, (2) multi-head attention, and (3) feed-forward networks. In batched processing, QKV generation and feed-forward networks involve compute-intensive matrix-matrix multiplications (GEMM), while multi-head attention requires bandwidth-heavy matrix-vector multiplications (GEMV). Machine learning accelerators like TPUs or NPUs are proficient in handling GEMM but are less efficient for GEMV computations. Conversely, Processing-in-Memory (PIM) technology is tailored for efficient GEMV computation, while it lacks the computational power to handle GEMM effectively.

Inspired by this insight, we propose NeuPIMs, a heterogeneous acceleration system that jointly exploits a conventional GEMM-focused NPU and GEMV-optimized PIM devices. The main challenge in efficiently integrating NPU and PIM lies in enabling concurrent operations on both platforms, each addressing a specific kernel type. First, existing PIMs typically operate in a "blocked" mode, allowing only either NPU or PIM to be active at any given time. Second, the inherent dependencies between GEMM and GEMV in LLMs restrict their parallel processing. To tackle these challenges, NeuPIMs is equipped with *dual row buffers* in each bank, facilitating the simultaneous management of memory read/write operations and PIM commands. Further, NeuPIMs employs a runtime *sub-batch interleaving* technique to maximize concurrent execution, leveraging batch parallelism to allow two independent sub-batches to be pipelined within a single NeuPIMs device. Our evaluation demonstrates that compared to GPU-only, NPU-only, and a naïve NPU+PIM integrated acceleration approaches, NeuPIMs achieves 3×, 2.4× and 1.6× throughput improvement, respectively.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651380

CCS Concepts: • **Computer systems organization** → **Parallel architectures**; *Neural networks*; **Heterogeneous (hybrid) systems**.

Keywords: Processing-in-memory (PIM), Neural processing unit (NPU), Heterogeneous system, Large language model (LLM), Inference serving, Transformer-based generative model (GPT)

ACM Reference Format:
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 *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620666.3651380

1 Introduction

Large Language Models (LLMs) are being widely deployed across various sectors such as natural language understanding [9, 12, 27, 67, 80, 85, 86], content generation [13, 57, 73, 74], and decision support [51]. However, a key challenge with these models is the substantial resource requirement they impose - both memory and compute. This paper specifically addresses the inference challenges in contemporary LLMs, with an emphasis on models like GPT4 [67] and LLaMA [80].

The algorithmic commonality of these state-of-the-art LLMs is that their model architecture constitutes a stack of decoder blocks. As illustrated in Figure 1(a), each block is structured around three primary layers: (1) Query-Key-Value (QKV) generation, (2) Multi-Head Attention (MHA), and (3) Feed-Forward Networks (FFNs). For efficient computation of these blocks, a prevalent strategy is batching multiple inference requests. Batching allows QKV generation and feed-forward layers to reuse weights across multiple requests, resulting in GEneral Matrix Multiplication (GEMM) operations between weight and activation matrices. Conversely, the multi-head attention layer requires multiplication between activation matrices and activation vectors with no data reuse opportunity, leading to GEneral Matrix-Vector Multiplication (GEMV) operations.

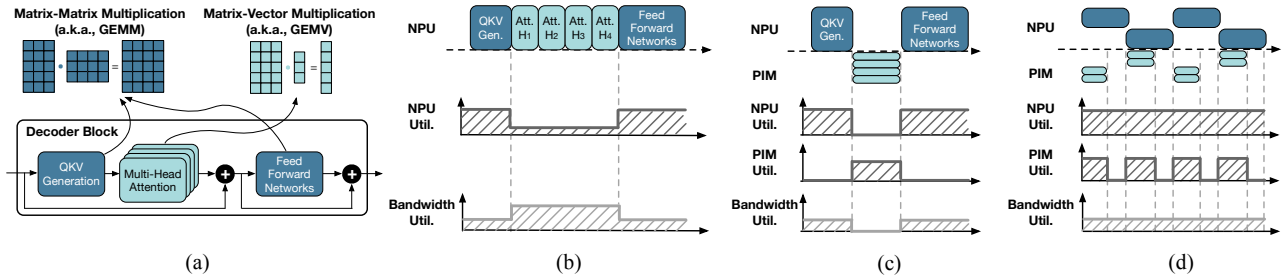


Figure 1. (a) Mathematical components of decoder blocks that constitute LLMs, (b) NPU-only baseline accelerator equipped with *non*-PIM memory (e.g., GPU), (c) NPU+PIM integrated baseline accelerator, and (d) the proposed NeuPIMs accelerator.

Overall, LLM inference involves the computation of numerous large-scale GEMMs and GEMVs. To address this computational demand, a common practice is to utilize high-performance machine learning (ML) accelerators, such as GPUs and TPUs. In this paper, we will refer to these ML accelerators as Neural Processing Units (NPUs). NPUs are often optimized for compute-intensive tasks, particularly for the efficient execution of GEMMs. However, their utility for GEMVs is less optimal due to the latter's lower arithmetic intensity, which leads to under-utilization of the NPU's computational resources. On the other hand, Processing-in-Memory (PIM) technology [6, 7, 10, 14–16, 19–26, 29–31, 33, 35–40, 43, 44, 46, 47, 49, 53, 59, 64, 68, 70, 82, 83, 88], while not as effective for GEMMs, shows promise for the bandwidth-intensive GEMV operations.

To this end, this work proposes NeuPIMs, a novel heterogeneous acceleration system for batched inference of LLMs. We architect NeuPIMs such that it effectively balances the utilization of memory bandwidth and computational resources of the system to improve the overall inference throughput. NeuPIMs jointly exploits (1) a conventional GEMM-centric NPU using a 2D cluster of multiple systolic arrays and (2) a multitude of GEMV-friendly processing-in-memory (PIM) accelerators. In designing NeuPIMs, we identify two major challenges:

- **Microarchitectural Challenge:** Current PIMs operate in a "blocked" mode, preventing the simultaneous execution of NPU and PIM. This serialization leads to an inherent under-utilization of resources.
- **Algorithmic Challenge:** In LLM decoder block, GEMM and GEMV operations have a data dependency. This algorithmic limitation fundamentally limits the possibility of parallel NPU+PIM computations.

NeuPIMs addresses the aforementioned challenges by taking a hardware-algorithm co-design approach and makes the following contributions:

(1) Microarchitectural Contribution: To facilitate NPU+PIM parallel execution, NeuPIMs introduces a modified PIM bank architecture that enables regular memory accesses to occur concurrently with GEMV operations within the PIM. This is achieved by employing distinct row buffers for these two functionalities, hereafter referred to as *dual row buffers*. Dual row buffers leverage the property of DRAM where multiple rows can be activated independently without affecting functionality. This further requires handling and scheduling of mixed commands for memory access and PIM operation at the memory controllers without violating DRAM timing parameters. To do so, NeuPIMs strategically intersperses the two types of commands, minimizing row activation delays. Additionally, a few composite commands are appended to the baseline PIM ISA, performing multiple GEMV operations and thereby amortizing the controlling cost.

(2) Algorithmic Contribution: To enable parallel executions of GEMM and GEMV operators within the decoder block, we introduce the *sub-batch interleaving* technique, which concurrently processes two sub-batch inference computations on the NeuPIMs system. As the two sub-batches are independent of each other, it is possible to parallelize the execution of GEMM operations from one sub-batch with GEMV operations from another sub-batch. This approach creates avenues for simultaneous executions, enhancing overall efficiency. With sub-batch interleaving, NeuPIMs aims to balance the workload between GEMM and GEMV operations effectively. To balance the pipeline of sub-batches, we estimate mappings from sequence lengths to the MHA execution latency on the PIM. This information allows NeuPIMs to partition a given batch such that it balances the total sum of sequence lengths in the sub-batches.

Combining the proposed microarchitectural and algorithmic innovations, NeuPIMs achieves high utilization on both NPU and PIM accelerators, thus offering significant throughput improvement over NPU-only and naïve NPU+PIM integrated baselines. Figure 1(b)-(d) visualizes the operator-accelerator mappings on NPU and/or PIM, along with their utilization trends for a short window in the execution runtime.

We evaluate the effectiveness of NeuPIMs using 4 variants of GPT3, a state-of-the-art LLM, with varying sizes. The evaluation utilizes real-world LLM inference datasets, ShareGPT and Alpaca, both accompanied by input and output sequence length information. We develop the NeuPIMs simulator[1] by integrating an open-source NPU simulator, ONNXim [3], with our in-house PIM simulator built on DRAMsim3 [50]. Our experimental results report that compared to an NPU-only and a naïve NPU+PIM integrated baseline accelerators, NeuPIMs

[1]Our simulator is available at https://github.com/casys-kaist/NeuPIMs.

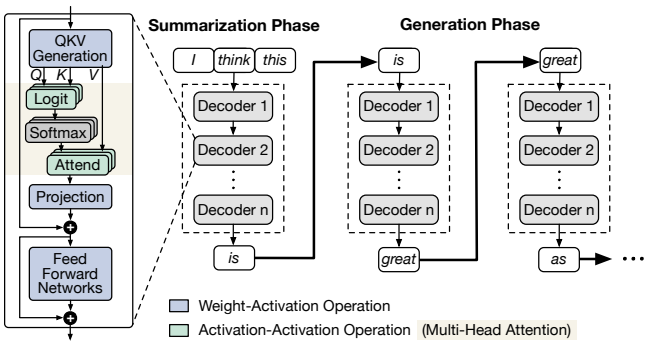
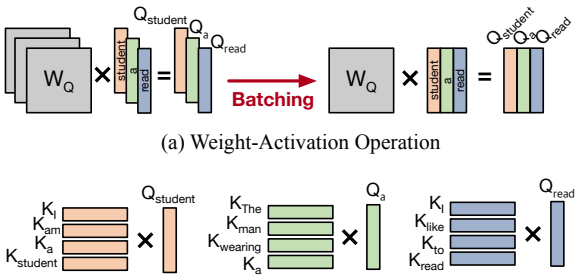


Figure 2. Model architecture and inference in LLMs.



(a) Weight-Activation Operation

(b) Activation-Activation Operation

Figure 3. Operators in a LLM decoder block.

achieves 2.4× and 1.6× throughput improvement, respectively. These significant throughput gains are attributed to the improved resource utilization of NPU and PIM from 28% and 17% to 65% and 26%, respectively. These compelling advantages highlight that NeuPIMs effectively overcome the limitations of existing solutions and take an effective initial step towards the practical deployment of PIM for LLM inference scenarios.

2 Background

2.1 Computational Characteristics of LLM Inference

Model architecture and execution of LLMs. Figure 2 illustrates the model architecture that all state-of-the-art large language models share [9, 18, 61, 78, 80, 86]. This illustration serves as a recurring example throughout the paper. For an input prompt (e.g., "I think this"), the model undergoes an summarization phase, encoding the input to establish context for the subsequent generation phase. In the generation phase, the model produces tokens one at each iteration in an autoregressive manner, using the generated key-value projections for the next iteration. Both phases constitute a sequence of decoder blocks, each comprising three major layers: (1) QKV generation, (2) multi-head attention (MHA), and (3) a set of feed-forward networks (FFNs).

Batched inference of LLMs. Computationally, the MHA layers have significantly different characteristics than QKV generation and FFN layers. Figure 3 denotes the example tensor operations of LLMs that show (a) weight-activation multiplications, and (b) activation-activation multiplications. The computations for QKV generation and FFNs are performed

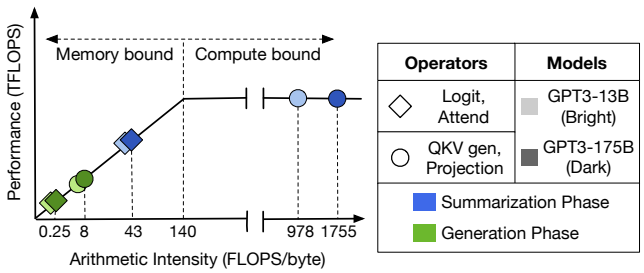


Figure 4. Arithmetic intensities of LLM layers.

by multiplying a per-token Q/K/V activation or attention vector with the trained weight matrices (GEMV). However, these GEMV operators are transformed into GEMMs when (1) they are located at the decoders in the summarization phase, getting multiple token vectors in parallel (e.g., "I think this") or (2) multiple inferences are batched, further parallelizing the computations for multiple single-token generation processes. On the other hand, the computations for MHA layers are multiplications between two different activations where one is for the current token (vector), and the other is for all the tokens before the current token (matrix), rendering a matrix-vector multiplication (GEMV). As the activation operands are unique for each inference request, their batching is not possible, making the computations highly memory bandwidth-bound.

Analysis of arithmetic intensity. To better understand the computational characteristics of LLM inference, we conduct a roofline analysis using two GPT3 variants, GPT3-13B and GPT3-175B. Figure 4 shows the relationship between the arithmetic intensity (FLOPS/byte) and performance (TFLOPS). We observe that for both models, the generation phases are severely memory-bound, while the summarization phases are compute-bound. As these two phases have algorithmic dependencies and occur alternately in a sequential manner, it is fundamentally challenging to achieve high resource utilization using a homogeneous computing platform. This insight motivates this work and drives us to design a heterogeneous system that combines a compute-centric systolic array-based NPU for GEMMs with memory-centric Processing-in-Memory (PIM) accelerators for GEMVs.

2.2 LLM Inference Serving

As there is a massive resource demand for LLMs, the de-facto practice is to build large-scale inference serving frameworks such as DeepSpeed [8], Orca [84] and vLLM [41]. These frameworks offer inference services for customer requests (i.e., prompts), which enables batching.

Selective batching. In general, batching is an effective method for neural network inference to improve resource utilization, while not sacrificing the latency requirement. However, MHA layers pose a challenge as they do not allow batching. This presents a difficulty for hyperscalers dealing with numerous customer requests while operating within limited compute resources. To address this, a recent work,

Orca [84], proposes a solution where attention layers are individually computed, while QKV generation and FFN layers are batched. This approach allows the system to still benefit from batching when possible; otherwise, it serializes the computation. This unique algorithmic property necessitates the simultaneous computations of GEMM and GEMV, which is the main motivation for this work.

Iteration level scheduling. Inference serving system receives requests in a streaming fashion without a deterministic schedule. Therefore, there is a need for a parallelization approach that can efficiently process the non-deterministically collected set of inference requests. Orca [84] additionally proposes to schedule batched inference at the beginning of every iteration. This allows new inference requests to be added to and terminated requests to be removed from the batch. Consequently, newly arrived requests do not need to wait until the generation phase for an already-started batch is terminated. This approach can significantly reduce the average latency for inference serving. NeuPIMs is built upon this scheduling technique, and thus, it manages the inference requests at the iteration boundaries.

Memory paging for attention. vLLM [41] is another recent effort to enhance the resource utilization of LLM inference serving systems, with a specific focus on memory management. As discussed in Section 2.1, the QKV generation layer produces KV cache, the input for the attention layers that can be reused in the generation phases. Leveraging this opportunity, LLM inference systems *cache* the KV projections in the memory, the size of which can be significant when the sequence length becomes large. vLLM introduces memory paging for this cached data, ensuring that a significant amount of memory is not pre-allocated long before its actual use. NeuPIMs employs the vLLM's paging technique, implementing the page-based memory allocation mechanism for KV cache, which effectively increases the batch size significantly.

It is worthwhile to note that NeuPIMs is designed to be deployed on an inference serving system that incorporates all of these aforementioned techniques.

3 Motivation

This section provides the motivation that underlies the design decisions of NeuPIMs. First, we identify problems of the existing GPU-based LLM inference serving systems, which motivates the NPU-PIM heterogeneous approach. Then, we will discuss the limitations of naïve NPU-PIM integration approach, defining the target research challenges of this work.

3.1 GPU-based LLM Inference Serving

Particularly for LLMs, as they require a lot of memory, it is a common practice to deploy them on a cluster of multiple GPUs [1, 2, 8, 17, 63], leveraging pipeline parallelism [28, 60] and/or tensor parallelism [61, 87].

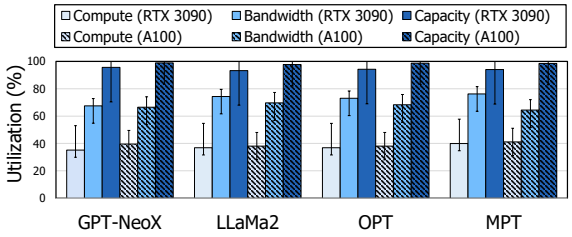


Figure 5. GPU resource utilization for four different LLMs.

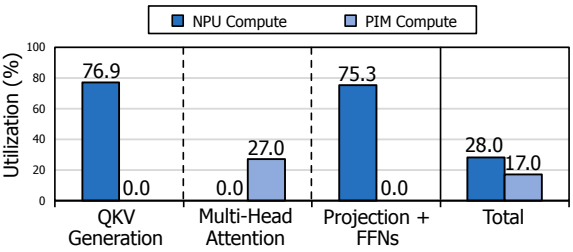


Figure 6. NPU-PIM resource utilization for decoder block.

Under-utilization of the GPU system. We analyze a GPU-equipped baseline system to understand the utilization of compute, memory, and bandwidth for LLM inference. We compare systems with NVIDIA GeForce RTX 3090 24GB and NVIDIA A100 40GB, running four different LLM models: GPT-NeoX, LLaMa2, OPT, and MPT. Figure 5 presents the utilization results along with the layer-wise variations as error bars. The figure illustrates that the capacity utilization closely approaches 100% despite the inherent imperfections in the parallelization schemes. This observation is intuitive as the number of GPUs used is determined based on the capacity constraints. However, the utilization of computational resources is consistently lower than 40%, which shows the cost-ineffectiveness of GPU-based LLM inference systems. This under-utilization is attributed to insufficient bandwidth, despite A100s being equipped with HBM, providing an aggregate of 1,555 GB/s. Unfortunately, this imbalance is inevitable as long as serial dependencies between GEMMs and GEMVs persist.

3.2 A Naïve NPU-PIM Approach

A straightforward approach to resolve this bandwidth bottleneck is to exploit the Processing-in-Memory (PIM) technology, which allows offloading the bandwidth-bound computations to its in-memory accelerator. Thus, we design a naïvely integrated NPU-PIM accelerator, exploiting a systolic array architecture for the NPU and incorporating a state-of-the-art PIM-based GEMV accelerator, Newton [25]. We use the same methodology as in Section 3.1, where we replace GPUs with a standard NPU-PIM integrated device. The detailed hardware and system simulation methodology are described in Section 8. Figure 6 presents the compute utilization of NPU and PIM for running different layers in the LLM decoder blocks. The results show that while NPU is busy running QKV generation, projection, and FFN layers, PIM utilization stays at zero. On the other hand, NPU utilization becomes almost

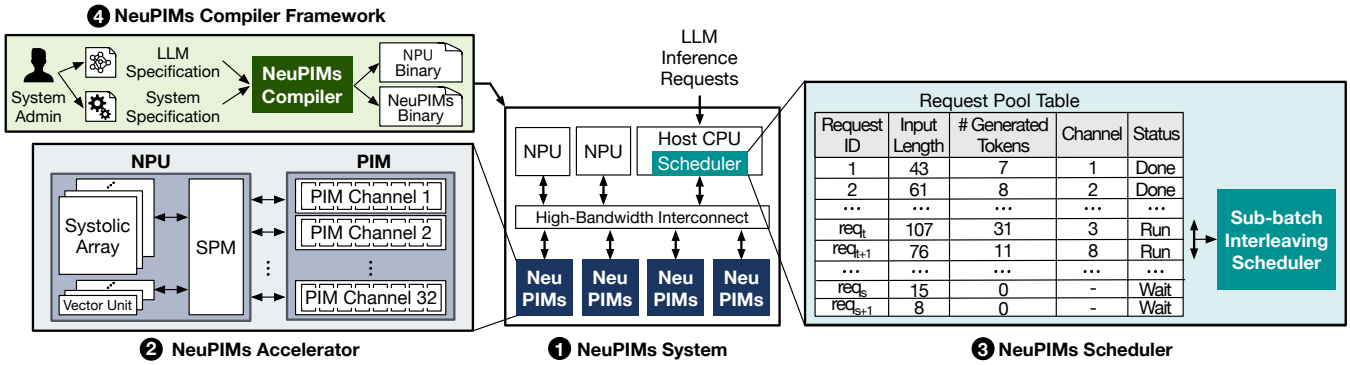


Figure 7. Overview of the proposed NeuPIMs system.

zero when PIM is running the MHA layers. Consequently, the combined utilization of NPU and PIM, when measured across the entire execution time, is less than 40% for both.

Necessity of concurrent NPU and PIM executions. The observed under-utilization is primarily due to the fundamental limitation in PIM's microarchitecture that disallows the concurrent execution of host (NPU) and PIM units, which serializes the disjoint resource usages. Consequently, the most critical challenge in realizing the practical use of PIM in NPU accelerators is enabling their parallel executions. This research problem constitutes the focus of this work.

4 Overview of NeuPIMs

Figure 7 illustrates the overview of the proposed NeuPIMs system. This system alleviates the low resource utilization of an LLM inference serving system. To achieve this goal, NeuPIMs comprises: (1) an NPU equipped with systolic arrays, vector processing units, and multiple HBM-based PIM channels that collaboratively process the batched inference requests, and (2) a scheduler that partitions an inference batch into two sub-batches and leverages sub-batch parallelism to enable their interleaved executions for enhanced NPU-PIM parallelization. Note that NPU in the NeuPIMs device is a general representation of any ML accelerator such as TPU [34] and is not the contribution of this work. In this paper, we propose a novel accelerator integrating PIM with NPU and the corresponding scheduling strategy. First, we provide an overview of the system.

❶ **NeuPIMs system.** This system comprises a host CPU, multiple NeuPIMs devices (i.e., NPU+PIM accelerators), and multiple standalone NPUs connected through a high-bandwidth interconnect such as PCIe and CXL. As the summarization phase is entirely composed of GEMMs, we delegate its computation to the standalone NPUs, while NeuPIMs focuses on the computation of the generation phase. While the diagram visualizes a single-node NeuPIMs system, the system can scale to multiple nodes, which will be discussed in more detail in Section 7. As in typical inference serving systems, our system receives the LLM inference requests in a streaming fashion. The requests are assigned to a PIM channel and queued in the request pool table until the on-going iteration is completed and a new iteration commences for the execution.

❷ **NeuPIMs accelerator.** We extend the design of standard PIM to support NeuPIMs LLM inferencing strategy. The bank architecture is expanded to include dual row buffers—one for PIM execution and the other for regular memory accesses, as illustrated in Figure 8. The dual row buffer architecture enables the NPU to perform memory read/write accesses on the bank rows that are not currently in use for PIM computations. This segregation is regulated by memory controllers to ensure that multiple activations are not issued over the same bank row (Section 5).

❸ **NeuPIMs scheduling algorithm.** Our prototype NeuPIMs accelerator has 32 HBM-based PIM channels, each of which is controlled by its own memory controller. The memory controllers manage the interleaving of memory read/write commands and PIM commands in a way that the inter-command timing delays are not violated, while maximizing the control path throughput. Effective interleaving is critical for performance since it directly affects the concurrent executions of NPU and PIM (Section 6).

❹ **NeuPIMs compiler framework.** NeuPIMs compiler framework is the frontend where the system admin provides the desired LLM and system specifications. The syntax of LLM specification largely resembles ONNX [58]. Upon receiving the specification, the compiler translates the model configuration into operations, each of which is represented as an intermediate representation (IR). For the given IR, our compiler produces NPU and NeuPIMs instruction binaries (i.e., Compute and MEM/PIM access instructions). This process involves adjusting tile sizes and the sequence of instructions to align with the NeuPIMs system specification.

5 NeuPIMs Architecture

5.1 PIM Microarchitecture for Concurrent Execution

Single row buffer for PIM-based accelerator. Figure 8(a) depicts the high-level architecture of PIM-based GEMV accelerators that utilize banks equipped with a single row buffer. For a GEMV, the vector operand is first located in the

global buffer shared across all banks in a channel. On the contrary, the rows of matrix operand are read from multiple banks simultaneously, exploiting bank-level parallelism, and located at their corresponding row buffers. When the operands are ready, the parallel multipliers and adder tree perform a partial dot-product by reading the broadcast vector input from the global buffer and per-bank row buffers.

Limitation of current PIM-based GEMV accelerators. Current PIM accelerators [25, 35, 49] operate in a "blocked" mode, preventing the simultaneous executions of NPU[2] and PIM. This limitation arises primarily because the memory bank utilizes a single row buffer that serves two purposes: read/write memory operations and PIM acceleration specifically for GEMV. These modes are managed sequentially, making it impossible to perform simultaneous executions. While this constraint does not significantly impact PIM system's performance for sole execution of GEMV, it becomes pertinent for LLM inferencing that requires both GEMM and GEMV operations. Addressing this issue, our work aims to facilitate the parallel execution of both modes within the PIM framework. This advancement is expected to significantly enhance performance in LLM inferencing applications, unlocking the full potential of PIM beyond its current limitations.

Extending PIM with dual row buffers. Figure 8(b) delineates the microarchitecture of NeuPIMs bank. The memory banks of NeuPIMs are equipped with *dual row buffers*, namely MEM row buffer and PIM row buffer, which are associated with two independent data paths. Memory (MEM) row buffer is exclusively used for regular memory read/write accesses, whereas PIM row buffer is employed for GEMV operation. In designing NeuPIMs, our design principle is to minimize the microarchitectural modification since the complication would impose significant area and power costs, lowering the practicality in the real-world system. Instead, we delegate the complications to the command interface and memory control mechanism, which will be elaborated below.

For prototyping and evaluating NeuPIMs, we choose an industry-developed PIM accelerator designed for GEMV, Newton [25]. However, note that the proposed techniques in this work are not bounded to the Newton architecture, rather applicable to any GEMV accelerator that follows the standard DRAM microarchitecture and command interface.

5.2 Memory Command Interface

Existing command interface for PIM-based GEMV. We develop the NeuPIMs device using a PIM accelerator with a modified command interface on top of the existing DRAM standard interface. There are four commands that collaboratively operate the PIM. First, to process GEMV in PIM banks, NeuPIMs must copy the operand vector to a global vector buffer shared by the banks within a channel.

[2]Henceforth in this paper, "NPU" refers exclusively to the device integrated within NeuPIMs, setting it apart from any standalone NPUs.

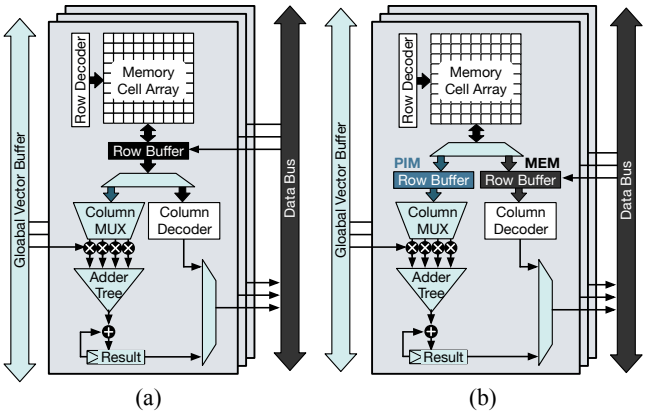


Figure 8. Microarchitecture of memory banks in (a) existing PIM accelerators with single row buffer banks, and (b) the proposed NeuPIMs with dual row buffer banks.

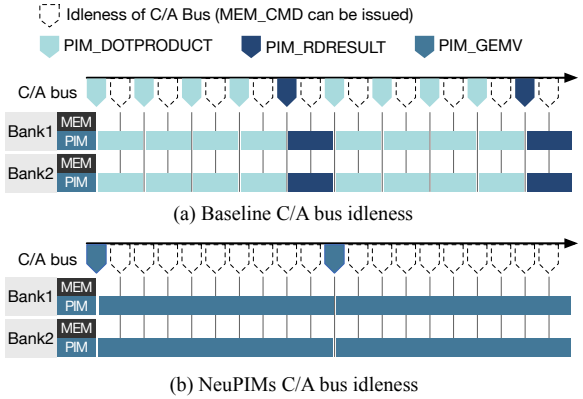


Figure 9. PIM command timing comparison.

Table 1. List of NeuPIMs commands.

Command	Description
PIM_HEADER	Configure a GEMV operation
PIM_GEMV	Perform k dot-products and read the results
PIM_PRECHARGE	Precharge PIM row buffer

NeuPIMs perform this operation using the PIM_GWRITE command, which copies a specific row of a particular bank to the global vector buffer. Similarly, it needs another command to activate the rows of the operand matrix into PIM row buffers across the banks. For this, the host produces grouped activation commands, called "PIM_ACTIVATION", which activate PIM row buffers of multiple banks simultaneously, usually for 4 banks at a time due to the power constraints (i.e., tFAW). Once all the banks are activated, the host sends "PIM_DOTPRODUCT" command that performs parallelized dot-product computation, extracting massively larger in-memory bandwidth than host-memory bandwidth. Finally, the "PIM_RDRESULT" command transfers the accumulated results to the host, ending a round of GEMV operation in PIM.

NeuPIMs command interface. We modify the command interface of baseline PIM by augmenting three additional commands that support new capabilities for NeuPIMs.

- **PIM_HEADER:** This command allows varying dimensionalities of GEMV operation. Existing PIM accelerators have a rigid architecture supporting a GEMV with fixed dimensionalities, and therefore, the GEMV's execution latency is deterministic. This property allows the memory controller to deterministically schedule the PIM commands without violating the DRAM refresh intervals. However, as NeuPIMs targets the GEMV operations of LLM's MHA layers, its dimensionality varies depending on the sequence length, and thus, the memory controller has no means to accurately calculate the latency, which may lead to DRAM refresh in the middle of PIM execution. To address this issue, NeuPIMs allows the software to initialize a GEMV execution by sending the PIM_HEADER command, which delivers the dimensionality information of scheduled GEMV operation. This way, the memory controller is able to estimate the end-to-end latency of GEMV operation and schedule its constituent commands without conflicting with the DRAM refresh.
- **PIM_GEMV:** The GEMV operation in existing PIM is controlled by a series of PIM_DOTPRODUCT commands, and the result is read using the PIM_RDRESULT commands, as depicted in Figure 9(a). This fine-grained control approach naturally results in substantial command traffic in the memory C/A bus. While this is not an issue for PIM that operates in a blocked mode, it becomes a significant concern for NeuPIMs that operate two functionalities in parallel. PIM_GEMV is a composite command that controls multiple dot-products simultaneously, and at the end, returns the result back to the host NPU. Figure 9(b) shows an example timeline that shows the reduced command traffic by PIM_GEMV. The number of dot-products, k, is given as an argument for the command.
- **PIM_PRECHARGE:** As the NeuPIMs banks have dual row buffers, there is a need for an additional command that specifically precharges the PIM row buffers once the GEMV is completed. PIM_PRECHARGE is the same as the regular PRECHARGE command except that it triggers precharge of the PIM row buffer.

5.3 Memory Controller

For NeuPIMs, we have multiple channels, each of which has multiple PIM banks. The LLM requests are assigned to one of these channels, with the MHA layer execution for each request being distributed across the channel's multiple PIM banks. The memory controllers located in their respective PIM channels are equipped with their individual PIM command queues. PIM commands are broadcast to all banks in the corresponding channel.

Interleaved scheduling of memory read/write and PIM commands. A challenge in the implementation of

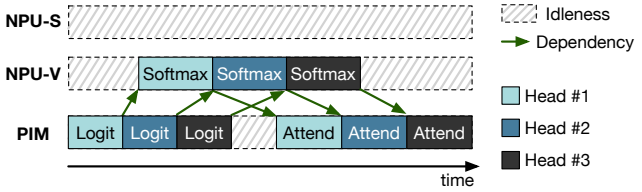


Figure 10. Overlapping opportunities of multi-head attention layers. NPU-S: systolic arrays; NPU-V: vector units.

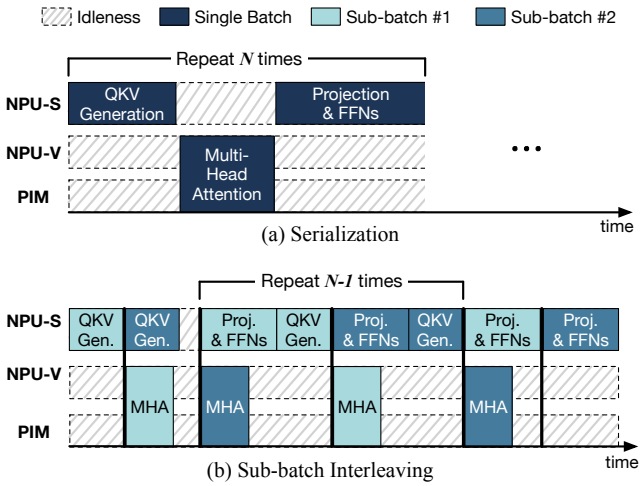


Figure 11. Example execution timelines of LLM decoder blocks: (a) Serialized execution, (b) Sub-batch interleaving. NPU-S: systolic arrays; NPU-V: vector units; N: the number of decoder blocks.

NeuPIMs memory controller is to interleave the memory read/write commands and PIM commands efficiently so that the command/address bus bandwidth does not become a performance bottleneck. NeuPIMs prioritize PIM commands over memory read/write commands, since the issuing delay of PIM commands is greater than that of memory commands, so the C/A bus bandwidth used to issue PIM commands is relatively small, allowing both commands to be issued without significant performance degradation.

6 NeuPIMs Scheduling

The integration of dual row buffers enables NeuPIMs to handle NPU's memory accesses and PIM commands simultaneously. This section describes computation overlapping opportunities for multi-head attention (MHA) layer execution and a novel request scheduling technique for batched LLM inferencing, dubbed *sub-batch interleaving*.

6.1 Overlapping Opportunities in MHA Layer

Figure 10 illustrates the overlapping of (1) logit and attend operations on PIM-side, and (2) softmax operations on NPU-side in NeuPIMs. As operations of multi-head attention can be decomposed to a head granularity, even naïve NPU-PIM integrated architecture should have an opportunity to harness

both resources to overlap operations. However, it cannot take advantage of this opportunity because operational results cannot be transferred between PIM units and vector units through PIM channels. In contrast, NeuPIMs, by employing dual row buffers, can concurrently harness both NPU and PIM. This configuration allows vector units within NeuPIMs to store partial logit (softmax) values without having to wait for the completion of the entire logit operations (GEMV) on the PIM, reducing the underutilization of integrated units.

It is worthwhile to note that this overlapping is only possible because there is head-level parallelism, which is only available within the multi-head attention layer. Further, the overlapping opportunity exists only between PIM and vector units of NPU, resulting in the NPU's systolic arrays largely remaining unused during the execution of the MHA layer.

6.2 Sub-batch Interleaving

Limitation of serialized executions. Figure 11(a) illustrates an example execution timeline of the decoder block operations, running on a naïve NPU-PIM integrated device. In particular, the figure depicts the algorithmic dependencies among QKV generation, multi-head attention, and projection & FFNs, within the batch. Due to the dependencies, they must be executed serially, inevitably leading to low utilization on both NPU and PIM.

Interleaving the two sub-batches. To tackle this challenge, we propose *sub-batch interleaving* that partitions one large batch into two sub-batches and alternate them to improve resource utilization. Figure 11(b) delineates the sub-batch interleaving technique that allows the simultaneous executions of PIM-friendly and NPU-friendly operations within the sub-batches on the PIM and NPU, respectively, significantly improving the utilization of both NPU and PIM.

Comparative analysis on the execution timelines. Let N denote the number of decoder blocks to be executed on a single NeuPIMs device. Figure 11(a) shows that without the use of sub-batch interleaving, the operators in each decoder block would be executed sequentially, leading to a total execution time equal to N times the per decoder-block execution time (i.e., QKV generation on NPU-S + MHA on PIM & NPU-V + Projection & FFNs on NPU-S).

However, as depicted in Figure 11(b), sub-batch interleaving effectively hides the execution times of MHA layer in the NPU-S execution times, resulting in the total execution time equal to (N - 1) times the per-sub-batch partial execution time (i.e., Projection, FFNs, and QKV generation on NPU-S) plus a single decoder-block execution time divided at the start and end. While the interleaving occurs, the NPU and PIM utilization is improved as their executions are effectively overlapped. Our empirical study demonstrates that the NeuPIMs execution time in the interleaved period is mostly bounded by the NPU execution time running GEMM operations, hiding the PIM execution time for MHA layer execution.

Algorithm 1: MHA Latency Estimation

Input: seq_len: *Sequence length of the request*
Parameter: E: *Model embedding size*
L_{tile}: *GEMV latency for one PIM tile*
L_{GWRITE}: *GWRITE latency for global buffer*
E: *Model embedding size*
P_{DRAM}: *DRAM page size*
B_{chnl}: *PIM banks per channel*
N_{head}: *Number of heads*
Output: L_{MHA}: *Estimated latency for MHA*

$L_{MHA} \leftarrow 0$;
/* ***GEMV latency for*** $Key^T \times Query$ */
$N_{tiles} \leftarrow (seq\_len \,/\, B_{chnl}) * (E \,/\, P_{DRAM})$;
L_{MHA} += $L_{GWRITE} * (E \,/\, P_{DRAM})$;
L_{MHA} += $L_{tile} * N_{tiles}$;
/* ***GEMV latency for*** $Logits \times Value$ */
$N_{tiles} \leftarrow ((E/N_{head})/B_{chnl}) * ((seq\_len/P_{DRAM}) * N_{head})$;
L_{MHA} += $L_{GWRITE} * ((seq\_len \,/\, P_{DRAM}) * N_{head})$;
L_{MHA} += $L_{tile} * N_{tiles}$;
return L_{MHA}

Challenges. For optimal exploitation of parallelism inherent in sub-batch interleaving, NeuPIMs must consider two key aspects: First, NeuPIMs requires a strategic balancing of the execution time for each sub-batch, particularly focusing on the multi-head attention. Since the latency of multi-head attention is determined by the channel processing the longest sequence, we must implement load balancing across the channels, ensuring an equitable distribution of token lengths. This challenge is addressed through a channel load balancing algorithm (Section 6.4). Second, NeuPIMs must ensure similar execution times for both sub-batches for efficient interleaved execution. The duration of each stage within the interleaving is bound by the processing time of more time-consuming sub-batch. For that, NeuPIMs introduces a sub-batch partitioning algorithm (Section 6.5).

6.3 Multi-Head Attention Latency Estimation

The latency of operations running in the NPU is largely dependent on the batch size of the inference. To apply optimization techniques for multi-head attention, we estimate the execution time of its operations by considering the key-value mapping to the PIM memory layout. Since the vector for GEMV is shared across the banks, the matrix for GEMV is interleaved row-wise to each banks. Consequently, key caches at the same row and column share the same layer and head index, with differing sequence indices. Conversely, value caches at the same row and column share the same layer, head, and sequence index, interleaving each head embedding into banks. Algorithm 1 takes this mapping into account to estimate the execution time of the multi-head attention latency.

6.4 Greedy Min-Load Bin Packing Algorithm.

To minimize the execution time discrepancy between the most congested channel and the load-free channel, we developed

Algorithm 2: Greedy Min-Load Bin Packing

Input: L_{req}: *A list for sqeunce length of new requests*
L_{chnl}: *A list for request allocation of channels*

```
/* Calculate each channel's total load by applying
   MHA latency estimation to each allocated request */
L_load ← [];
foreach chnl in L_chnl do
    Sum_load ← 0;
    foreach req in chnl do
        Sum_load += MHALatencyEstimation(req)
    end
    L_load.append(Sum_load);
end
/* Allocate each request by greedy algorithm */
foreach new_req in L_req do
    min_index = Min(L_load).index();
    L_chnl[min_index].append(new_req);
    load_req = MHALatencyEstimation(new_req);
    L_load[min_index] += load_req;
end
return L_chnl
```

Algorithm 3: Sub-Batch Partitioning

Input: L_{req}: *A list of active request set in each channel*
Output: SB_1, SB_2: *Sub-batchs for interleaving*

```
turn ← True;
SB_1, SB_2 ← [];
foreach req_chnl in L_req do
    bsize ← Size(req_chnl) / 2;
    if Size(req_chnl) % 2 != 0 then
        bsize = turn ? ceil(bsize) : floor(bsize);
        turn = !turn;
    end
    bsize = int(bsize);
    SB_1.append(req_chnl[: bsize]);
    SB_2.append(req_chnl[bsize :]);
end
return SB_1, SB_2
```

the channel load balancing algorithm. Algorithm 2 leverages the aforementioned multi-head attention latency estimation to batch requests. It initially sorts the batch of requests in decreasing order of input token length. Then, NeuPIMs places a request with the longest token length in the channel with minimal load. At each iteration, it updates the estimated latency, taking into account the newly appended request.

NeuPIMs allocate LLM inference requests to one of the PIM channels. A PIM channel constitutes multiple PIM banks, each of which partially executes the MHA layers of the assigned requests. Thus, it is crucial to minimize the execution time discrepancy between the most congested channel and the load-free channel for load-balancing purposes. To this end, we develop *greedy min-load bin packing algorithm*, as presented in Algorithm 2. This algorithm leverages the aforementioned multi-head attention latency estimation to batch requests. As implied by its name, the algorithm greedily allocates requests starting from the longest sequence length to the least loaded PIM channel sequentially. Therefore, it initially sorts the batch of requests in decreasing order of input token length. Then, NeuPIMs places a request with the longest token length in the channel with minimal load. At each iteration, it updates the estimated latency, taking into account the newly appended request.

6.5 Sub-batch Partitioning Algorithm

Given the dependency of NPU-friendly operations on the batch size of inference, it is essential to maintain a balanced size between the two sub-batches. As outlined in Algorithm 3, the approach involves dividing the requests for each channel into halves and appending each half to one of the sub-batches.

7 Scaling NeuPIMs System

Model parallelism, which involves partitioning model parameters across multiple NeuPIMs devices for parallel processing, is essential for inference tasks of LLM, due to the limited memory capacity of NeuPIMs devices. In this section, we will discuss the applicability of NeuPIMs systems to pipeline parallelism and tensor parallelism, two widely used model parallelism techniques in deep learning libraries [1, 61, 62] for LLM inference. Note that, these model parallelism strategies are not the contribution of this work, but this section highlights the adaptability of such techniques to NeuPIMs system.

7.1 Pipeline Parallelism of NeuPIMs System

Pipeline parallelism involves dividing the model layer-wise, so that several layers of model are placed on a single NeuPIMs device. To facilitate parallel processing, the batch is divided into micro-batches corresponding to the pipeline depth, and each device processes them in a pipelined manner. This approach can also be applied to NeuPIMs systems in a similar manner.

When applying pipeline parallelism to the NeuPIMs systems, the number of decoder blocks executed on a single NeuPIMs device decreases proportionally. As mentioned in Section 6.2, the reduced decoder blocks co-located in one NeuPIMs device would lower the achievable performance benefits. Furthermore, exploiting pipeline parallelism leads to smaller batch size. Using the sub-batch interleaving technique further reduces batch size, which would lead to under-utilization of the NPU systolic arrays.

7.2 Tensor Parallelism of NeuPIMs System

Tensor parallelism is a parallelization approach that splits the model tensors into multiple shards and distributes them across devices. Multiple devices execute on their respective model shards in parallel, the results of which are aggregated at the end of the step. This aggregation requires communication between devices.

Table 2. NeuPIMs hardware specification.

NPU Configuration		HBM Organization	
Systolic Arrays/Chip	8	Banks/Bankgroup	4
Vector Units/Chip	8	Banks/Channel	32
HBM Channels/Chip	32	Frequency	1GHz
Systolic Array Size	128 x 128	Capacity/Channel	1GB
Vector Unit Size	128 x 1	Page Size	1KB
HBM Timing Parameter			
tRP = 14, tRCD = 14, tRAS = 34, tRRD_L = 6, tWR = 16, tCCD_S = 1, tCCD_L = 2, tREFI = 3900, tRFC = 260, tFAW = 30			

Although the sub-batch interleaving technique increases communication frequency twofold, the total communication traffic remains unchanged compared to the non-partitioned batch, resulting in only a modest overhead from communication. Moreover, the sub-batch that completes first among the two sub-batches can engage in communication, while the other sub-batch performs computations. This further reduces the communication latency, mitigating the increased communication overhead.

Inspired by the aforementioned insights, we prioritize the exploitation of tensor parallelism over pipeline parallelism and start employing pipeline parallelism when the model size is too large to exclusively leverage tensor parallelism.

8 Evaluation

8.1 Methodology

Baseline. For evaluation, we compare three baseline systems along with our NeuPIMs system, namely GPU-only, NPU-only, and NPU+PIM.

- **GPU-only:** GPU-only is a real GPU system. We conduct experiments with an A100 40GB GPU, and we compile the LLM batched inference workload with PyTorch [71].
- **NPU-only:** NPU-only represents existing NPU accelerators such as TPU, without any PIM capabilities. We assume that this baseline has the equivalent memory bandwidth with other alternatives for fairness. Moreover, a NPU is equipped with not only systolic array but also GPU-like vector processing units to support *non*-GEMM operators.
- **NPU+PIM:** NPU+PIM is also a PIM-enabled NPU baseline, which integrates existing PIM-based GEMV accelerators [25] with the off-the-shelf NPU accelerator. We merely map the GEMV operations in MHA layers to the PIM, while all the others are computed at the NPU side. We allocate the requests to PIM channels in a round-robin manner.

Cycle-level simulation. We build the NeuPIMs simulator on two open-source cycle-accurate simulators, ONNXim [3] and DRAMsim3 [50]. We link the two simulators by modifying the memory interface of ONNXim and offloading the memory accesses to the PIM simulator based on DRAMsim3.

Hardware specifications. We prototype NeuPIMs accelerator using a set of hardware specifications, which are listed in Table 2. Our NeuPIMs accelerator prototype is a multi-chiplet design containing 8 systolic arrays, each integrated with a SIMD vector unit that serves activation operations. Each memory channel controls 32 PIM banks, which offer in aggregate 1GB memory capacity. Note that while we choose this set of specifications for prototyping purposes, the NeuPIMs architecture is orthogonal to these choices, allowing varying configurations depending on the model size and data-specific properties (e.g., sequence lengths).

Table 3. The evaluated LLM configurations.

Model	# Layers	# Heads	d_{model}	# TP	# PP
GPT3-7B	32	32	4096	4	1
GPT3-13B	40	40	5120	4	1
GPT3-30B	48	56	7168	4	2
GPT3-175B	96	96	12288	8	4

LLM models. We use four variants of GPT3, a state-of-the-art LLM developed by OpenAI as described in Table 3. While our experiments focus on GPT-3 model variants, NeuPIMs can host any decoder-based generation models, offering generality and wide applicability.

Datasets. We use two real-world LLM inferencing datasets, ShareGPT [79] and Alpaca [76]. ShareGPT dataset is a set of conversations scraped from the real-world user log of ChatGPT [66]. Alpaca dataset is an instruction dataset generated by OpenAI's text-davinci-003 engine. The two datasets have distributions for input and output sequences. ShareGPT has an average input token length of 80 and an output of 296, while Alpaca has shorter sequence lengths of 12 and 56 for input and output, respectively.

Workload. As running experiments for inference serving scenarios with a cycle-accurate simulator is infeasible, we develop an alternative methodology to synthesize workloads for system-level evaluation. To define the search space for workloads, we consider various hyperparameters, including model types, batch sizes, and tensor/pipeline parallelism. For each permutation of these hyperparameters, we simulate the inference serving for a fixed amount of time, randomly picking sequence lengths from the datasets. This way, we can warm up the inference batch in a way that the batch is filled with requests having various sequence lengths. We sample 10 different batches and use them to measure the throughputs of different hyperparameter combinations.

8.2 Results

Throughput. Figure 12 reports the throughput comparison results between the three baselines and NeuPIMs. The GPU-only and NPU-only baseline systems show marginal differences since they both execute end-to-end decoder block operations without PIM, including bandwidth-bound multi-head attention. Simply integrating PIM with the NPU, i.e., NPU+PIM already offers, on average 1.5× throughput improvement compared to the NPU-only baseline because the bandwidth-bound

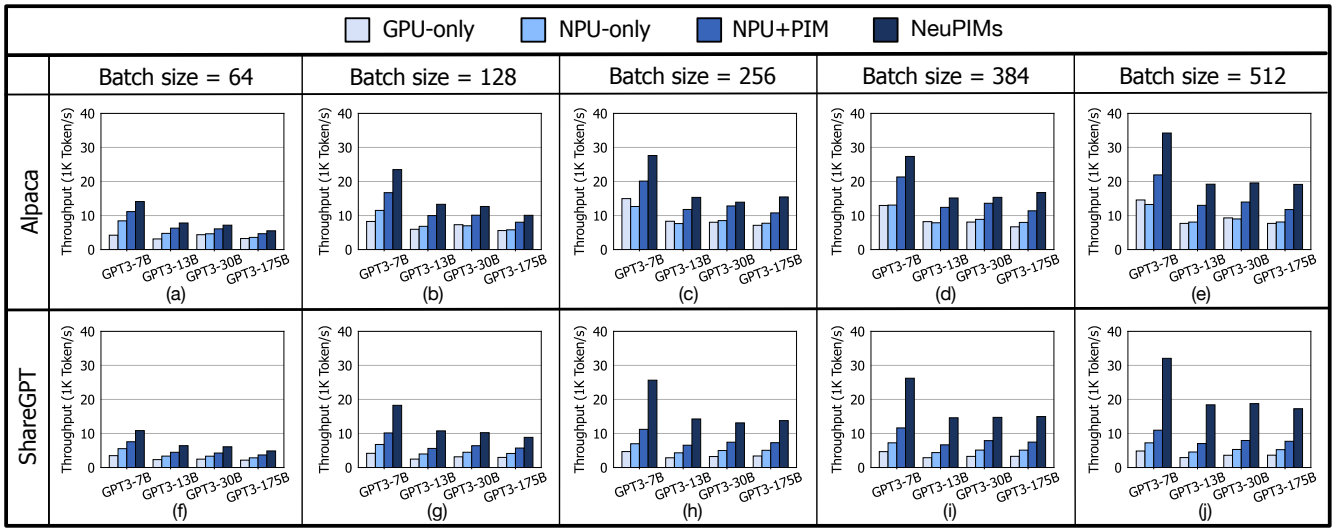


Figure 12. Throughput comparison results of GPU-only, NPU-only, NPU+PIM, and NeuPIMs. We use the two datasets, (a) Alpaca and (b) ShareGPT, using the batch sizes including 64, 128, 256, 384, and 512.

Table 4. Average resource utilization of NPU/PIM compute resource and memory bandwidth utilization.

	NPU-only	NPU+PIM	NeuPIMS
NPU	12.3%	28.0%	**64.9%**
PIM	-	17.0%	**26.4%**
Bandwidth	67.6%	27.4%	**85.4%**

GEMV operations in MHA are all offloaded to the PIM. However, NeuPIMs consistently surpasses the NPU+PIM baseline, and offers *additional* throughput improvements over the NPU+PIM baseline across all the models and datasets, ranging from 13% to 3×. These trends are observed consistently for both datasets, with larger gains observed for ShareGPT, given its longer input/output sequences, offering increased acceleration opportunities for PIMs. Furthermore, as the batch size increases from 64 to 512, the throughput gains exhibit substantial growth. This is because the NeuPIMs system effectively shifts the bottleneck from bandwidth to compute towards the NPU, thus allowing NeuPIMs to extract higher performance from batched computation as the batch size grows. Note that NeuPIMs achieves significant throughput improvements using the same memory capacity, which demonstrates its cost-effectiveness, especially in a datacenter-scale inference serving scenario where larger batch sizes are advantageous.

Utilization. The main source of large throughput gain is the improved resource utilization across the board. Table 4 compares the average utilization for three major resources in the baselines and NeuPIMs, when using GPT3-30B, the batch size of 256, and the ShareGPT dataset. As NPU+PIM offloads the bandwidth-bound MHA operations to PIM, it increases NPU utilization by 28.0%. However, NPU+PIM still suffers from temporal blocking due to the GEMM-GEMV dependencies in LLM inferencing. NeuPIMs overcomes this limitation, achieving 64.9% and 26.4% utilization on NPU and PIM, respectively, through the concurrent NPU+PIM execution capability. We observe similar trends from the other system configurations, which demonstrate the effectiveness of the proposed technique in enabling concurrent NPU+PIM execution by NeuPIMs.

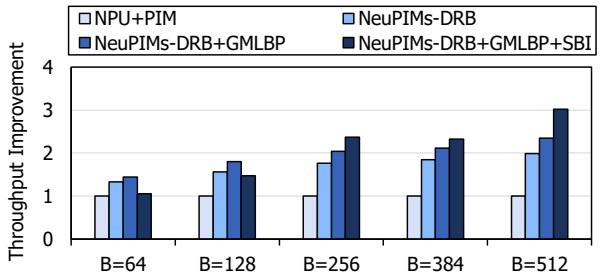


Figure 13. We use the GPT3-7B model and ShareGPT dataset for this experiment. DRB: Dual Row Buffers; GMLBP: Greedy Min-Load Bin Packing algorithm; SBI: Sub-Batch Interleaving.

Ablation study. Using NPU+PIM as the baseline, we augment the proposed three techniques and observe the performance behaviors, shown in Figure 13. For all batch sizes, the dual row buffers offer 69.7% throughput improvement on average, which has the largest impact on the performance as it enables concurrent NPU+PIM execution without significant overhead. For the channel loading balancing, our greedy min-load bin packing algorithm also always offers performance benefits by evenly distributing the requests to the available channels. In contrast to the previous two techniques, the sub-batch interleaving technique does not always yield gains. For small batch sizes, partitioning the batch into two may cause underutilization in a NPU systolic array, leading to inefficiency, and the penalty from pipelining

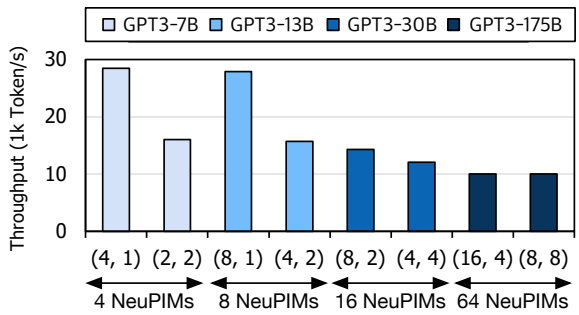


Figure 14. Throughput of multi-NeuPIMs system as the parallelization schemes change. We employ four combinations of tensor parallelism (TP) and pipeline parallelism (PP), represented as (TP, PP).

Table 5. NeuPIMs power overhead.

Baseline	Average Power	
NPU-only	HBM (non-PIM)	364.1 mW
NeuPIMs	Dual row buffered PIM	634.8 mW

could outweigh the benefits. However, when the batch size is equal to or larger than 256, NeuPIMs achieve the highest throughput, suggesting that batched inference with a large batch size is preferable for the NeuPIMs-based system.

Implication of parallelization schemes. As the LLM size increases, the NeuPIMs system must scale the number of devices to harness tensor and pipeline parallelisms. Figure 14 analyzes the implications of such parallelization schemes on the system throughput. For the experiment, we fix the total number of requests to 256, while the batch size per device varies depending on the parallelization scheme. The results highlight the preference for exploiting tensor parallelism over the pipeline counterpart, as it maintains a large batch size, resulting in better efficiency at the NPU. We observe this trend consistently for all model variants, while the overall throughput decreases since the per-device batch size becomes small, due to low NPU utilization.

Area overhead. The main source of area overhead in the NeuPIMs architecture is the dual row buffer. We use CACTI 7.0 [11] with 22nm technology to measure the area overhead by doubling the row buffer resource usage in the tool configuration. We observe 3.11% area overhead, which is marginal considering the significant performance boost provided the microarchitectural addition.

Power overhead. Compared to the NPU-only counterpart, NeuPIMs requires higher power in memory because it operates NPU and PIM concurrently. We examine this power implication by measuring it using Micron's DRAM power model [32] provided by DRAMsim3 [50]. We assume all-bank computation command incurs 4× higher power than read command [25]. Moreover, the "additional" row buffer requires DRAM to consume more background power to hold each row buffer status. We measure the total power overhead

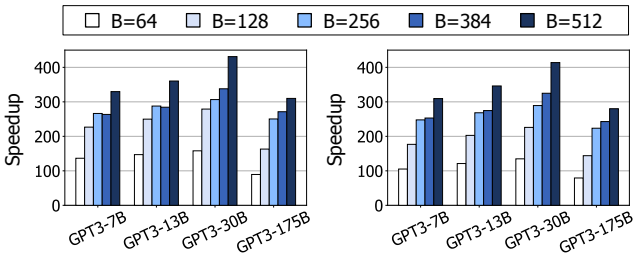


Figure 15. Speedup of NeuPIMs over TransPIM [89].

by aggregating all these factors together. Table 5 compares the average power of NeuPIMs and NPU-only system where NPUs are equipped with vanilla HBMs. NeuPIMs exhibit 1.8× higher power consumption, offering 2.4× speedup, which can be translated into 25% energy reduction.

Comparison with TransPIM. TransPIM [89] is a standalone PIM-only solution that operates the entirety of transformer operators within PIM devices. As there is no open-source simulator for TransPIM, we develop our own TransPIM simulator based on DRAMsim3 [50]. We align the memory specifications of TransPIM, such as HBM timing parameters and capacity, with those used for NeuPIMs and the NPU+PIM baseline.

Figure 15 reports the speedup of NeuPIMs over TransPIM [89] NeuPIMs shows an average 228× higher throughput than TransPIM. The significant performance gap is attributed to the effectiveness of GEMM computation executed on the NPU in the case of NeuPIMs, as opposed to PIM in TransPIM. That is, TransPIM specifically targets single-batch transformer model inference, making it unsuitable for batched inference. Additionally, we observe that the token-based dataflow and ring-broadcast mechanism proposed in TransPIM are designed to target encoder block operations, making them inefficient for decoder-based LLM inference. Overall, NeuPIMs consistently deliver superior performance compared to TransPIM, achieving speedups ranging from 79× to 431×.

9 Discussion

Model training. As training has fixed-length sequences for both input and output, it entirely entails GEMMs, not requiring any GEMVs for its computations. PIM targets accelerating bandwidth-bounded GEMV operations, delivering significantly poorer performance for GEMMs. Therefore, while a NeuPIMs system can be used for training, its efficiency is limited.

Integration with production software stack. As described in Section 4, the NeuPIMs compiler framework employs a similar interface with the modern machine learning libraries such as ONNX, PyTorch, and JAX. Therefore, the integration of NeuPIMs solution with the existing production software stack would require one to write a translator, which can convert the ONNX-, PyTorch-, and Jax-defined model representations into our LLM specification. With the translator being developed, the rest of NeuPIMs system would remain

the same since it is already a holistic system stack that constitutes an inference serving scheduler, operator compilers for NPU and PIM, and an inference execution runtime.

10 Related Work

LLM inference serving. Multiple LLM serving systems optimize for their inference performance either through reducing the memory footprint [42, 81], improving the kernel execution strategy [87], determining the partitioning techniques for intra- and inter- operator [61, 72, 77] execution, or a combination of these [2, 8, 41, 52, 56, 62, 63, 65, 84]. In this work, we specifically tackle the utilization of the current hardware platforms which deploy these models through a compute and I/O suitable platforms, NPUs and PIMs, to create a more efficient system. Moreover, to ensure such a heterogeneous system can perform well for LLM inferencing, we offer a scheduling policy. Prior works that support kernel optimizations for better utilization of GPUs for LLMs, cannot fully mitigate the I/O and bandwidth bottleneck of GEMV kernels. Instead in this work, we build a system that can benefit from existing optimizations such as selective batching, KV caching, etc. to offer better utilization across the transformer model architecture.

PIM for language model support. TransPIM [89] is a PIM solution that accelerates the end-to-end transformer inference using PIM. The work proposes a data loading overhead reduction technique by customizing its dataflow for transformer models. TransPIM is optimized for attention operations of transformer encoder blocks, making it unsuitable for LLM inference based on decoder blocks. Furthermore, it is tailored for single-request inference, offering suboptimal performance for batched inference scenarios. AttAcc [16] further offers an accelerator (with PIM) for attention layer to reduce the data movement for the KV matrices. Instead NeuPIMs proposes a new system for PIM accelerator in addition to the scheduling of operations for the end to end inference of LLMs.

There are also variety of prior works that leverage PIM for GEMV operations [25, 40, 44, 48, 49, 68, 83] due to their inherent potential in benefits towards bandwidth bound applications. However, none of these works enable simultaneous execution of PIM and NPU operations, necessary for the efficient execution of LLM inference.

Heterogeneous acceleration pipeline for deep learning. There are a variety of prior works that propose a pipelined solution for machine learning [4, 55, 69, 75], however none of these prior works leverage PIM to alleviate the bandwidth requirement of LLMs. Certain prior works use an accelerator for specific models [5, 45, 54], however, they do not alleviate the under-utilization of GEMV and GEMM operations in transformer decoder blocks.

11 Conclusion

Large Language Model (LLM) inferencing, given its significance, demands dedicated resources that can be deployed at scale. However, these models present a confluence of challenges, encompassing high memory capacity, high compute intensity, and bandwidth constraints. In this work we propose a novel system, NeuPIMs, that integrates NPU (a general ML accelerator) with PIM technology to mitigate the limitations associated with different operations and their dataflow in the transformer layers. We introduce a novel scheduling and execution strategy for the proposed system, that can better utilize HBM memory, compute intensive NPU, and the PIM accelerator for LLM inference serving. Results indicate that the system developed in this work offers a 1.6× throughput improvement compared to the baseline system that naïvely integrates an NPU with the PIM accelerator.

Acknowledgments

We thank our shepherd Vidushi Goyal and the anonymous reviewers for their comments and feedback. This research is supported by Institute of Information & communications Technology Planning & Evaluation (IITP) (No.2022-0-01037, No.2018-0-00503), Information Technology Research Center (ITRC) support program (IITP-2024-2020-0-01795), and Artificial Intelligence Graduate School Program (KAIST) (No.2019-0-00075), funded by the Korea government (MSIT).

References

[1] Nvidia Tensor RT 4.0. https://developer.nvidia.com/tensorrt.
[2] HuggingFace. https://github.com/huggingface/transformers/tree/main, 2022.
[3] ONNXim NPU Simulator. https://github.com/PSAL-POSTECH/ONNXim, 2024.
[4] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. Accelerating Recommendation System Training by Leveraging Popular Choices. *Proc. VLDB Endow.*, 15(1):127–140, jan 2022.
[5] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. Heterogeneous Acceleration Pipeline for Recommendation System Training. *arXiv preprint arXiv:2204.05436*, 2022.
[6] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*, 2015.
[7] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *ISCA*, 2015.
[8] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. Technical report, Microsoft, 06 2022.
[9] Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernandez Abrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark

Díaz, Nan Du, Ethan Dyer, Vlad Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, Guy Gur-Ari, Steven Hand, Hadi Hashemi, Le Hou, Joshua Howland, Andrea Hu, Jeffrey Hui, Jeremy Hurwitz, Michael Isard, Abe Ittycheriah, Matthew Jagielski, Wenhao Jia, Kathleen Kenealy, Maxim Krikun, Sneha Kudugunta, Chang Lan, Katherine Lee, Benjamin Lee, Eric Li, Music Li, Wei Li, YaGuang Li, Jian Li, Hyeontaek Lim, Hanzhao Lin, Zhongtao Liu, Frederick Liu, Marcello Maggioni, Aroma Mahendru, Joshua Maynez, Vedant Misra, Maysam Moussalem, Zachary Nado, John Nham, Eric Ni, Andrew Nystrom, Alicia Parrish, Marie Pellat, Martin Polacek, Alex Polozov, Reiner Pope, Siyuan Qiao, Emily Reif, Bryan Richter, Parker Riley, Alex Castro Ros, Aurko Roy, Brennan Saeta, Rajkumar Samuel, Renee Shelby, Ambrose Slone, Daniel Smilkov, David R. So, Daniel Sohn, Simon Tokumine, Dasha Valter, Vijay Vasudevan, Kiran Vodrahalli, Xuezhi Wang, Pidong Wang, Zirui Wang, Tao Wang, John Wieting, Yuhuai Wu, Kelvin Xu, Yunhan Xu, Linting Xue, Pengcheng Yin, Jiahui Yu, Qiao Zhang, Steven Zheng, Ce Zheng, Weikang Zhou, Denny Zhou, Slav Petrov, and Yonghui Wu. PaLM 2 Technical Report, 2023.

[10] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems. In *MICRO*, 2016.

[11] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2), jun 2017.

[12] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. GPT-NeoX-20B: An Open-Source Autoregressive Language Model, 2022.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, 2021.

[14] Shuang Chen, Yi Jiang, Christina Delimitrou, and José F. Martínez. PIMCloud: QoS-Aware Resource Management of Latency-Critical Applications in Clouds with Processing-in-Memory. In *HPCA*, 2022.

[15] Benjamin Y. Cho, Jeageun Jung, and Mattan Erez. Accelerating bandwidth-bound deep learning inference with main-memory accelerators. In *SC*, 2021.

[16] Jaewan Choi, Jaehyun Park, Kwanhee Kyung, Nam Sung Kim, and Jung Ho Ahn. Unleashing the potential of pim: Accelerating large batched inference of transformer-based generative models. *IEEE Computer Architecture Letters*, 22(2):113–116, 2023.

[17] ONNX Runtime developers. ONNX Runtime. https://onnxruntime.ai/, 2021.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *arXiv*, 2018.

[19] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. In *MICRO*, 2019.

[20] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures. *SIGMETRICS Perform. Eval. Rev.*, 50(1):33–34, jul 2022.

[21] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. In *Abstract Proceedings of the 2022 ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2022.

[22] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. SparseP: Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. In *ISVLSI*, 2022.

[23] Juan Gómez-Luna, Yuxin Guo, Sylvan Brocard, Julien Legriel, Remy Cimadomo, Geraldo F. Oliveira, Gagandeep Singh, and Onur Mutlu. Machine Learning Training on a Real Processing-in-Memory System. In *ISVLSI*, 2022.

[24] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access*, 10:52565–52608, 2022.

[25] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and T. N. Vijaykumar. Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning. In *MICRO*, 2020.

[26] Jaehoon Heo, Yongwon Shin, Sangjin Choi, Sungwoong Yune, Jung-Hoon Kim, Hyojin Sung, Youngjin Kwon, and Joo-Young Kim. PRIMO: A Full-Stack Processing-in-DRAM Emulation Framework for Machine Learning Workloads. In *ICCAD*, 2023.

[27] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training Compute-Optimal Large Language Models, 2022.

[28] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*, 2019.

[29] Bongjoon Hyun, Taehun Kim, Dongjae Lee, and Minsoo Rhu. Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology. In *HPCA*, 2024.

[30] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision. In *ISCA*, 2019.

[31] Mohsen Imani, Saikishan Pampana, Saransh Gupta, Minxuan Zhou, Yeseong Kim, and Tajana Rosing. DUAL: Acceleration of Clustering Algorithms using Digital-based Processing In-Memory. In *MICRO*, 2020.

[32] J Janzen. Calculating memory system power for DDR3. *Micron Designline*, 13(1), 2008.

[33] Gilbert Jonatan, Haeyoon Cho, Hyojun Son, Xiangyu Wu, Neal Livesay, Evelio Mora, Kaustubh Shivdikar, José L. Abellán, Ajay Joshi, David Kaeli, and John Kim. Scalability Limitations of Processing-in-Memory using Real System Evaluations. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2024.

[34] 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. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *ISCA*, 2023.

[35] Hongju Kal, Chanyoung Yoo, and Won Woo Ro. AESPA: Asynchronous Execution Scheme to Exploit Bank-Level Parallelism of Processing-in-Memory. In *MICRO*, 2023.

[36] Shinhaeng Kang, Sukhan Lee, Byeongho Kim, Hweesoo Kim, Kyomin Sohn, Nam Sung Kim, and Eojin Lee. An FPGA-based RNN-T Inference Accelerator with PIM-HBM. In *FPGA*, 2022.

[37] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 790–803, 2020.

[38] Donghyeon Kim, Taehoon Kim, Inyong Hwang, Taehyeong Park, Hanjun Kim, Youngsok Kim, and Yongjun Park. Virtual PIM: Resource-Aware Dynamic DPU Allocation and Workload Scheduling Framework for Multi-DPU PIM Architecture. In *PACT*, 2023.

[39] Heesu Kim, Hanmin Park, Taehyun Kim, Kwanheum Cho, Eojin Lee, Soojung Ryu, Hyuk-Jae Lee, Kiyoung Choi, and Jinho Lee. GradPIM: A Practical Processing-in-DRAM Architecture for Gradient Descent. In *HPCA*, 2021.

[40] Jin Hyun Kim, Shin-haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Bengseng Phuah, Jihyun Choi, Jinin So, YeonGon Cho, JoonHo Song, Jangseok Choi, Jeonghyeon Cho, Kyomin Sohn, Youngsoo Sohn, Kwangil Park, and Nam Sung Kim. Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond. In *Hot Chips*, 2021.

[41] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP*, 2023.

[42] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP*, 2023.

[43] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyun-Sung Shin, Jin Kim, BengSeng Phuah, HyoungMin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, SooYoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, JoonHo Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung 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 *ISSCC*, 2021.

[44] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *MICRO*, 2019.

[45] Youngeun Kwon and Minsoo Rhu. Training Personalized Recommendation Systems from (GPU) Scratch: Look Forward Not Backwards. In *ISCA*, 2022.

[46] Ann Franchesca Laguna, Arman Kazemi, Michael Niemier, and X. Sharon Hu. In-Memory Computing based Accelerator for Transformer Networks for Long Sequences. In *DATE*, 2021.

[47] Juhyoung Lee, Jihoon Kim, Wooyoung Jo, Sangyeob Kim, Sangjin Kim, Donghyeon Han, Jinsu Lee, and Hoi-Jun Yoo. An Energy-efficient Floating-Point DNN Processor using Heterogeneous Computing Architecture with Exponent-Computing-in-Memory. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021.

[48] Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Kornijcuk Vladimir, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Jaewook Lee, Donguc Ko, Younggun Jun, Keewon Cho, Ilwoong Kim, Choungki Song, Chunseok Jeong, Daehan Kwon, Jieun Jang, Il Park, Junhyun Chun, and Joohwan 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 *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, 2022.

[49] 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. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In *ISCA*, 2021.

[50] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020.

[51] Shuang Li, Xavier Puig, Chris Paxton, Yilun Du, Clinton Wang, Linxi Fan, Tao Chen, De-An Huang, Ekin Akyürek, Anima Anandkumar, et al. Pre-trained language models for interactive decision-making. *Advances in Neural Information Processing Systems*, 35:31199–31212, 2022.

[52] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *OSDI*, 2023.

[53] Haifeng Liu, Long Zheng, Yu Huang, Chaoqiang Liu, Xiangyu Ye, Jingrui Yuan, Xiaofei Liao, Hai Jin, and Jingling Xue. Accelerating Personalized Recommendation with Cross-level Near-Memory Processing. In *ISCA*, 2023.

[54] Divya Mahajan, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. TABLA: A Unified Template-based Framework for Accelerating Statistical Machine Learning. In *HPCA*, 2016.

[55] Divya Mahajan, Joon Kyung Kim, Jacob Sacks, Adel Ardalan, Arun Kumar, and Hadi Esmaeilzadeh. In-RDBMS Hardware Acceleration of Advanced Analytics. In *PVLDB*, 2018.

[56] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. *arXiv preprint arXiv:2311.15566*, 2023.

[57] Microsoft. Github copilot. https://github.com/features/copilot, 2022.

[58] Facebook Research Microsoft. Onnx: an open format to represent deep learning models. http://onnx.ai/, 2017.

[59] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *HPCA*, 2017.

[60] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *SOSP*, 2019.

[61] Nvidia. Megatron-lm. https://github.com/NVIDIA/Megatron-LM.

[62] Nvidia. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM.

[63] NVIDIA. NVIDIA Triton. https://developer.nvidia.com/triton-inference-server, 2020.

[64] Geraldo F. Oliveira, Juan Gómez-Luna, Saugata Ghose, Amirali Boroumand, and Onur Mutlu. Accelerating neural network inference with processing-in-dram: From the edge to the cloud. *IEEE Micro*, 42(6):25–38, 2022.

[65] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. TensorFlow-Serving: Flexible, High-Performance ML Serving. *CoRR*, abs/1712.06139, 2017.

[66] OpenAI. chatgpt. https://chatgpt.com/blog/chatgpt, 2023.

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

[68] Jaehyun Park, Byeongho Kim, Sungmin Yun, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory. In *MICRO*, 2021.

[69] Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. Scale-out acceleration for machine learning. In *MICRO*, October 2016.

[70] Sang-Soo Park, KyungSoo Kim, Jinin So, Jin Jung, Jonggeon Lee, Kyoungwan Woo, Nayeon Kim, Younghyun Lee, Hyungyo Kim, Yongsuk Kwon,

Jinhyun Kim, Jieun Lee, YeonGon Cho, Yongmin Tai, Jeonghyeon Cho, Hoyoung Song, Jung Ho Ahn, and Nam Sung Kim. An LPDDR-based CXL-PNM Platform for TCO-Efficient GPT Inference. In *HPCA*, 2024.

[71] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library, 2019.

[72] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently Scaling Transformer Inference, 2022.

[73] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical Text-Conditional Image Generation with CLIP Latents, 2022.

[74] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code, 2023.

[75] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Misra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *MICRO*, October 2016.

[76] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[77] Jakub M Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of dnn graph operators. *Advances in Neural Information Processing Systems*, 33, 2020.

[78] MosaicML NLP Team. Introducing MPT-30B: Raising the bar for open-source foundation models, 2023.

[79] ShareGPT Team. ShareGPT. https://sharegpt.com, 2023.

[80] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[81] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. LightSeq: A High Performance Inference Library for Transformers, 2021.

[82] Zhengrong Wang, Christopher Liu, Aman Arora, Lizy John, and Tony Nowatzki. Infinity Stream: Portable and Programmer-Friendly In-/Near-Memory Fusion. In *ASPLOS*, 2023.

[83] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *HPCA*, 2021.

[84] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *OSDI*, 2022.

[85] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, Zixuan Ma, Yufei Xue, Jidong Zhai, Wenguang Chen, Peng Zhang, Yuxiao Dong, and Jie Tang. GLM-130B: An Open Bilingual Pre-trained Model, 2023.

[86] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models, 2022.

[87] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, Joseph Gonzalez, and Ion Stoica. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *OSDI*, 2022.

[88] Minxuan Zhou, Guoyang Chen, Mohsen Imani, Saransh Gupta, Weifeng Zhang, and Tajana Rosing. PIM-DL: Boosting DNN Inference on Digital Processing In-Memory Architectures via Data Layout Optimizations. In *PACT*, 2021.

[89] Minxuan Zhou, Weihong Xu, Jaeyoung Kang, and Tajana Rosing. TransPIM: A Memory-based Acceleration via Software-Hardware Co-Design for Transformer. In *HPCA*, 2022.