SpecPIM: Accelerating Speculative Inference on PIM-Enabled System via Architecture-Dataflow Co-Exploration

Cong Li
School of Integrated Circuits
Peking University
Beijing, China
leesou@pku.edu.cn

Zhe Zhou
School of Integrated Circuits
School of Computer Science
Peking University
Beijing, China
zhou.zhe@pku.edu.cn

Size Zheng
School of Integrated Circuits
School of Computer Science
Peking University
Beijing, China
zhengsz@pku.edu.cn

Jiaxi Zhang
School of Computer Science
Peking University
Beijing, China
zhangjiaxi@pku.edu.cn

Yun Liang
School of Integrated Circuits
Peking University
Beijing Advanced Innovation Center for Integrated Circuits
Beijing, China
ericlyun@pku.edu.cn

Guangyu Sun*
School of Integrated Circuits
Peking University
Beijing Advanced Innovation Center for Integrated Circuits
Beijing, China
gsun@pku.edu.cn

Abstract

Generative large language models' (LLMs) inference suffers from inefficiency because of the token dependency brought by autoregressive decoding. Recently, speculative inference has been proposed to alleviate this problem, which introduces small language models to generate draft tokens and adopts the original large language model to conduct verification. Although speculative inference can enhance the efficiency of the decoding procedure, we find that it presents variable resource demands due to the distinct computation patterns of the models used in speculative inference. This variability impedes the full realization of speculative inference's acceleration potential in current systems.

To tackle this problem, we propose SpecPIM to accelerate speculative inference on the PIM-enabled system. SpecPIM aims to boost the performance of speculative inference by extensively exploring the heterogeneity brought by both the algorithm and the architecture. To this end, we construct the architecture design space to satisfy each model's disparate resource demands and dedicate the dataflow design space to fully utilize the system's hardware resources. Based on the co-design space, we propose a design space exploration (DSE) framework to provide the optimal design under different target scenarios. Compared with speculative inference on GPUs and existing PIM-based LLM accelerators, SpecPIM achieves 1.52×/2.02× geomean speedup and 6.67×/2.68× geomean higher energy efficiency.

CCS Concepts: • **Computer systems organization → Heterogeneous (hybrid) systems**.

Keywords: near-memory processing, large language models, speculative inference, domain-specific accelerator

ACM Reference Format:
Cong Li, Zhe Zhou, Size Zheng, Jiaxi Zhang, Yun Liang, and Guangyu Sun. 2024. SpecPIM: Accelerating Speculative Inference on PIM-Enabled System via Architecture-Dataflow Co-Exploration. 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/10.1145/3620666.3651352

*Corresponding author.

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

ACM ISBN 979-8-4007-0386-7/24/04…$15.00
https://doi.org/10.1145/3620666.3651352

1 Introduction

In recent years, generative large language models (LLMs) have made great breakthroughs in the field of artificial intelligence. Many LLMs such as GPT-family [6, 49, 52], OPT [70], LLaMA-family [62, 63], PaLM [10] and other variants [54, 61], have demonstrated extraordinary ability to solve a wide range of neural language processing (NLP) tasks, such as chatbot [17, 48], code generation [8, 16], machine translation [54, 70], question reasoning [10, 70], etc. Such capability makes LLM inference an indispensable workload for many cloud companies [2, 16, 17, 48].

LLM's inference can be divided into two key stages: prefill and decoding. In the prefill stage, the LLM takes an initial sequence of tokens (known as "prompt") as input and processes it in a single step. After processing, the LLM generates the first output token. Moving on to the decoding stage, the LLM takes the previously generated tokens as input and conducts autoregressive decoding: The LLM produces one new token per iteration, gradually building up the output sequence. Such a procedure introduces the dependency between adjacent tokens and substantially inflates the number of decoding iterations, creating a significant burden of computation and data movement. Consequently, autoregressive decoding emerges as a severe bottleneck in LLM inference. Previous work has shown that given the output token length of 128, the decoding stage of GPT-3 175B takes up more than 88% latency even when the prompt length is 2048 [9].

To alleviate such inefficiency, the emerging speculative inference technique [7, 24, 38, 43] becomes a promising solution. The key insight is that there are "easier" inference steps that can be handled by less powerful models. Therefore, speculative inference introduces smaller language models, named draft language models (DLMs), to generate draft tokens, and adopts the original model, also called target language model (TLM), to verify the correctness of these tokens. In this way, speculative inference can generate tokens that are equivalent to TLM's autoregressive decoding outputs. Besides, it can also achieve higher decoding efficiency because DLMs incur much lighter overhead than the TLM, and the TLM is invoked less frequently than autoregressive decoding.

Speculative inference introduces disparate computation patterns between the TLM and the DLMs: The TLM processes multiple tokens in parallel during verification, while the DLMs still conduct autoregressive decoding and generate tokens one at a time. Such heterogeneity not only results in the variation in hardware resource demands of different models but also makes it vital to accelerate both computation-intensive and memory-intensive operators for speculative inference. However, current GPU-based systems fall short of efficiently handling memory-intensive operators. Although the DRAM-based Processing-in-Memory (DRAM-PIM) technique has been introduced to accelerate memory-bound operators in LLM inference [9, 31, 33], existing PIM-based LLM accelerators only target for *single LLM*'s acceleration. Their heterogeneity-agnostic architecture design and the under-explored dataflow design hinder them from achieving the full acceleration potential of speculative inference.

To tackle this problem, we propose SpecPIM to accelerate speculative inference on the PIM-enabled system with the awareness of such heterogeneity. SpecPIM is the first work aiming to comprehensively explore both the algorithmic heterogeneity inherent in speculative inference and the architectural heterogeneity between centralized computing and DRAM-PIMs. To this end, we first construct SpecPIM's architecture design space by adjusting PIM resource allocation to satisfy each model's resource demands. We then propose SpecPIM's dataflow design space to optimize the multi-model collaboration workflow together with the efficiency of single-model execution. Based on the co-design space, we propose a design space exploration (DSE) framework to find out the optimal design for different scenarios. To summarize, we have made the following main contributions:

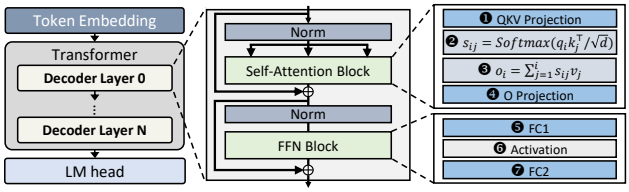

Figure 1. Decoder-based LLM Structure.

- We analyze the characteristics of speculative inference and the disparate computation patterns of different models in speculative inference.
- We construct SpecPIM's architecture design space and dataflow design space by comprehensively considering the algorithmic and architectural heterogeneity.
- We propose SpecPIM's DSE framework to find out the optimal design under different scenarios.

Experiments on various speculative inference tasks demonstrate that SpecPIM achieves 1.52× speedup and 6.67× higher energy efficiency compared with speculative inference on GPUs (geomean). Compared with speculative inference on existing PIM-based LLM accelerators, SpecPIM achieves 2.02× speedup and 2.68× higher energy efficiency (geomean).

2 Background

2.1 Transformer-based LLMs

Mainstream LLMs are built atop transformer decoder layers [64]. As illustrated in Figure 1, a decoder-based LLM comprises a token embedding, a series of decoder layers, and a language model (LM) head. The token embedding converts input tokens to embeddings which decoder layers can process, and the LM head generates new tokens according to the outputs of the last transformer layer. A classical transformer decoder layer contains a self-attention block and a feed-forward network (FFN) block, both of which are accompanied by normalization and residual operators. The self-attention block takes a sequence of hidden states $X = (x_1, ..., x_n) \in \mathbb{R}^{n \times d}$ as input ($n > 1$ in prefill stage, and $n = 1$ in decoding stage). These vectors are first projected to query (q_i), key (k_i), and value (v_i) vectors by fully connected (FC) layers (❶). Then, these vectors are split into H heads. In each head, every query vector is multiplied with the key vectors before its current position to get attention scores (❷). These scores are then applied to corresponding value vectors

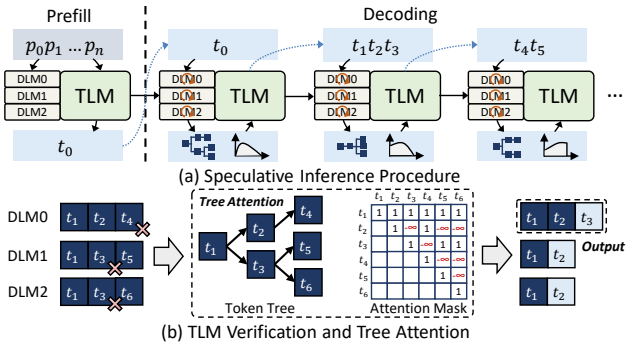

Figure 2. Overview of Speculative Inference.

to get weighted average vectors (❸). All weighted average outputs of each token are concatenated and projected by another FC layer (❹). Since the self-attention requires the key and value vectors of *all previous tokens*, they are usually stored during the whole generation procedure to avoid re-computation, which is known as *KV Cache*.

In classical transformers, the FFN block follows behind the self-attention block, which contains two FC layers (❺❼). The first layer's output needs to be processed by an activation function before sending it to the second FC layer (❻). In several LLM variants, the FFN block's structure or position relative to the self-attention block can be different from the classical design. For example, LLaMA [62, 63] introduces one more FC layer to the FFN module:

$$FFN(X) = FC_3(FC_1(X) \odot Swish(FC_2(X))) \quad (1)$$

PaLM [10] further adjusts the classical "serialized" network structure to the "parallel" structure listed below:

$$Y = X + FFN(Norm(X)) + Attention(Norm(X)) \quad (2)$$

2.2 Speculative Inference

Speculative inference's procedure is shown in Figure 2-(a). In the prefill stage, all models process the prompt and the TLM generates the first token. In each iteration of the decoding stage, all DLMs first generate tokens autoregressively. After each DLM generates K ($K > 1$) tokens, the TLM collects these tokens and their generation probabilities and then begins verification. For t_i in a generated sequence $t_1, ..., t_K$, assume the DLM's probability is $D(t_i)$ and the TLM's probability is $T(t_i)$. If we have: $\frac{T(t_i)}{D(t_i)} < r_i$, where $r_i \sim U(0, 1)$, the TLM will reject t_i and re-sample the correct token. For example, in Figure 2-(b), t_4 in DLM0 and t_3 in DLM1/DLM2 are rejected, so the TLM re-samples t_3 for DLM0 and t_2 for DLM1/DLM2 to correct the generation. This rule enables speculative inference to get results equivalent to TLM's autoregressive decoding [38]. The insight is that $\frac{T(t_i)}{D(t_i)} < 1$ means the TLM has lower confidence in t_i. If the confidence gap is lower than a threshold, we should reject t_i. The longest sequence after verification is the output of the current iteration.

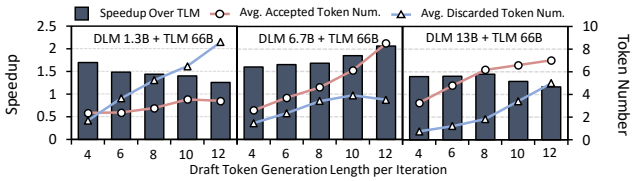

Figure 3. Analysis of DLM's Generation Length.

Since there are duplicate tokens in the sequences generated by different DLMs, speculative inference adopts *tree attention* mechanism [43] to avoid re-computation. As illustrated in Figure 2-(b), all sequences can be merged into a tree structure. If we send the token tree as a single sequence to the TLM, we will get the wrong results because the precedents on the tree path are just a subset of precedents in the compressed sequence. To reflect the correct token dependency, we need to adjust the mask in self-attention. For example, since t_5's precedents in the token tree only include t_1, t_3, we need to mask t_2, t_4 to get the correct results. In this way, tree attention keeps the correctness and does not change the computation pattern of self-attention.

3 Motivation

3.1 Analysis of Speculative Inference

We begin our motivational analysis by revealing the characteristics of speculative inference. Specifically, we select OPT [70] 66B as the TLM and select the DLM from OPT 1.3B/6.7B/13B. All models use FP16 datatype. The inference batch size is 1, and both of the input/output token lengths are 128. We equip one DLM with the TLM under all settings and conduct all experiments on four 40GB A100 GPUs.

Effect of Verification Length: We first examine the speedup of speculative inference when changing the token number verified by the TLM in each iteration. As illustrated in figure 3, we can find that speculative inference can achieve 1.17×-2.06× speedup under all settings. Besides, we can also find that increasing DLM's token generation length in each iteration brings disparate effects on speculative inference's performance under different settings, which is because of the following trade-off: On the one hand, more tokens can be accepted on average in each iteration. In this way, we can use fewer iterations to finish the text generation, thus achieving higher speedup. On the other hand, the average discarded token number also increases, which results in generating more redundant tokens, and thus hinders further speedup. Therefore, to achieve the best speedup for speculative inference, we need to choose a reasonable token generation length to balance this trade-off.

Arithmetic Property Analysis: To better understand the disparate arithmetic property between the TLM and the DLMs, we take the generation of one sequence as an example

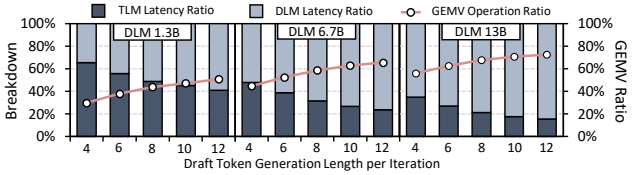

Figure 4. Speculative Inference Latency Breakdown.

to analyze the arithmetic intensity (AI) of different operators. For FC operators (i.e., ❶❹❺❼ in Figure 1), assume the input shape and the weight shape are $L_{in} \times d_{in}$ and $d_{in} \times d_{out}$, where L_{in}, d_{in}, d_{out} indicate the input token number, input vector length, and output vector length, respectively. For DLMs, we have $L_{in} = 1$ since it conducts autoregressive decoding. Therefore, FC operators in DLMs conduct low-AI (~1 MAC/Byte) GEMV operations. On the other hand, for the TLM, we have $L_{in} > 1$ because it processes the token tree as a single sequence. Since TLM's L_{in} is typically 2-10 [38], while its d_{out} is typically larger than 1000, the AI of TLM's FC operators can be derived as $\frac{L_{in} \times d_{in} \times d_{out}}{L_{in} \times d_{in} + d_{in} \times d_{out}} = \frac{L_{in} \times d_{out}}{L_{in} + d_{out}} \approx L_{in}$, which is larger than 1 MAC/Byte. Therefore, TLM's FC operators behave more computation-intensive than that of DLMs.

For Attention operators (i.e., ❷❸ in Figure 1), the GEMM operation in each attention head can also be expressed as the multiplication between a $L_{in} \times d_{in}$ input and a $d_{in} \times d_{out}$ K/V matrix. We also have $L_{in} = 1$ for DLMs and $L_{in} > 1$ for the TLM. Therefore, the GEMM operation in each attention head shares similar computation properties with the FC operators (i.e., the TLM is more computation-intensive than the DLMs).

Speculative Inference Latency Breakdown: To further analyze the bottleneck of speculative inference, we break down speculative inference's latency between the TLM and the DLM. As illustrated in Figure 4, we can find that the latency proportion of each model varies under different settings. Accordingly, the bottleneck of speculative inference changes between the computation-intensive operators and the memory-intensive operators because the TLM and the DLM behave in different computation patterns as analyzed above. To further reveal this phenomenon, we choose the GEMV as the most representative memory-bound operator in our motivational experiments and profile its latency proportion. As Figure 4 shows, the ratio of GEMV operators increases along with DLM's latency proportion. Accordingly, the bottleneck in speculative inference also switches to memory-intensive operators. Therefore, it is vital to handle both the computation-intensive and the memory-intensive operators to fully accelerate speculative inference.

However, existing GPU-based systems fall short of efficiently handling the memory-bound operators. Although we can enlarge the inference batch size to improve data reuse and reduce memory access intensity, the batch size cannot always scale too large because of the following reasons: First, the execution latency grows linearly with the batch size [3, 57]. Large batch size may violate the tight latency service level objective (SLO) under the online serving scenario [40]. Second, pipeline parallelism is a commonly used parallelism strategy for LLMs [3, 20, 40, 44, 71], which splits different decoder layers into multiple workers and executes the model in a pipeline manner. To improve the pipeline utilization, we need to split the original batch into multiple smaller micro-batches [3, 44, 71], which also shrinks the batched query number per processing. Considering GEMM operation under small batches is still memory-bound for GPUs [3], we need more effective approaches to accelerate the memory-intensive operators in speculative inference.

Table 1. Summary of Existing PIM-based LLM Accelerators

Name	Centralized Processor	PIM Type	Target LLM	PIM Operator
MI100-PIM GPU [31] (Samsung)	AMD MI100 GPU	HBM-PIM	GPT-J 6B	FC
AiM-Centric Card [33] (SK-Hynix)	Xilinx VU9P FPGA	GDDR-PIM	GPT-3 6.7B/20B	FC & Attention
AttAcc [9]	xPU (GPU, TPU, etc.)	HBM-PIM	GPT-3 175B	Attention

3.2 Chances and Challenges of DRAM-PIMs

Recently, DRAM-based Processing-in-Memory (DRAM-PIM) technique has received more and more attention [1, 4, 5, 11–14, 18, 19, 21, 27–31, 33–37, 39, 41, 45, 51, 55, 65, 66, 68, 69, 75, 76] to accelerate memory-bound operators. By incorporating processing engines (PEs) into memory modules, DRAM-PIMs can not only provide 8× higher memory bandwidth than normal memory but also greatly reduce the data-movement energy consumption [33, 36]. Therefore, accelerators equipped with HBM/GDDR-based DRAM-PIMs have been proposed from both the academia [9] and the industry [31, 33] to handle the memory-bound operators in LLM inference. As summarized in Table 1, Samsung equips HBM-PIM cubes with AMD's MI100 GPU [31] and offloads all FC layers to PIM to accelerate the single-batch GPT-J 6B inference task. AttAcc [9] utilizes HBM-PIM to hold the KV Cache and accelerate attention operators. SK-Hynix offloads both FC and Attention operators to GDDR6-PIM accelerator card named AiM [33], which can outperform A100 GPUs on the single-batch GPT-3 6.7B/20B inference task.

However, existing DRAM-PIM-based LLM accelerators only target accelerating *single LLM*'s inference, while speculative inference involves the interaction among multiple models with disparate computation patterns. To unleash the full potential of DRAM-PIMs for speculative inference acceleration, we still need to tackle the following challenges:

C1: ***The architecture design should meet the disparate resource demands in speculative inference.*** Current PIM-based LLM accelerators equip fixed PIM resources to all devices. For example, the HBM-PIM used in MI100-PIM GPU and AttAcc equips four PIM dies to each HBM cube [36], while the GDDR-PIM adopted by SK-Hynix's AiM-centric card equips PIM PEs to all GDDR6 packages [33]. However, fixed resource allocation does not always meet the varied

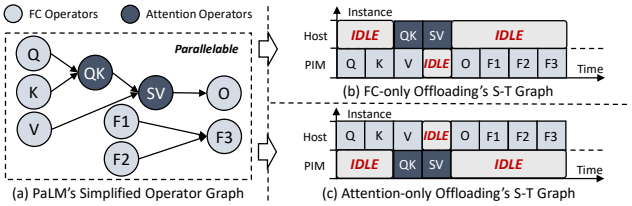

Figure 5. Illustration of Fixed Operator Mapping Strategies.

resource requirements in speculative inference. This inadequacy can be analyzed from two aspects:

On the one hand, as discussed in Section 3.1, the distinct computation patterns between the TLM and the DLMs give rise to diverse demands for PIM resources. Moreover, the computation intensity of different models is influenced by factors such as batch size, verification length, etc. Therefore, the most suitable resource allocation for each model varies across different scenarios, necessitating an exploration of optimal architectural designs to accommodate such variability.

On the other hand, integrating PIM PEs into memory channels compromises memory capacity due to PIM PE's area overhead. For example, Samsung's HBM-PIM [36] and SK-Hynix's AiM [33] halve the memory capacity to accommodate PIM PEs. Although we can gain much higher bandwidth for memory-intensive operators by equipping more PIM resources, the memory capacity of each device diminishes. This could entail the need for more devices to hold each model considering the huge memory capacity requirements of LLMs. Therefore, it is essential to explore the most suitable PIM resource allocation for each model, striking a balance between capacity and bandwidth.

C2: *The dataflow should be further optimized to efficiently utilize the system's hardware resources.* The PIM-enabled system aware of model characteristics contains two aspects of architecture heterogeneity: (1) Different models need to be executed on distinct devices so that we can explore the hardware resource allocation. (2) Each model's execution needs to be arranged between the centralized computation engine and the PIM PEs. Such heterogeneity poses the following challenges to the system's dataflow design:

Multi-Model Collaboration: The interdependence between the TLM and the DLMs results in severe resource under-utilization given that different models are executed on distinct devices. As described in Section 2.2, the TLM and DLMs require each other's outputs for processing. Therefore, when these models are waiting for new outputs, their corresponding devices have to be idle, which leads to significant waste in the system's hardware resources. How to utilize idle resources to boost speculative inference's performance is a challenging issue with a lack of discussion.

Single-Model Execution: The execution dataflow between the centralized host processor and the DRAM-PIMs is still under-explored. On the one hand, as listed in Table 1, existing DRAM-PIM-based LLM accelerators adopt *fixed operator offloading strategies*, which can lead to the following deficiencies: (1) Fixed offloading strategies are agnostic to PIM's memory capacity. If the data volume of PIM operators exceeds PIM's memory capacity, we have to store PIM operators' data in normal memory and transfer them to the PIM memory repeatedly, which can severely hurt the system's performance. (2) Fixed offloading strategies cannot always fit the disparate computation patterns of models in speculative inference as discussed above. (3) Fixed offloading strategies cannot fully utilize the inter-operator parallelism provided by the model architecture. For example, as illustrated in Figure 5, PaLM's parallel transformer layer provides abundant parallelism opportunities among operators. If we use fixed offloading strategies such as FC/Attention-only offloading, most operators have to be executed sequentially, and thus we cannot fully utilize the parallelism opportunity between the host processor and the PIM PEs. Although operator mapping has been extensively explored on homogeneous spatial architectures in previous works [15, 23, 26, 32, 42, 50, 67, 72–74], how to describe and explore operator mapping strategies on heterogeneous PIM-enabled systems to overcome these drawbacks is still lack of discussion.

On the other hand, PIM operators are typically executed on multiple PEs because existing DRAM-PIM designs typically place PEs near a few memory banks [33, 36], whose capacity cannot hold the whole PIM operator. It is necessary to explore the trade-offs brought by different tiling strategies so that we can allocate computation tasks more reasonably.

To tackle these challenges, we propose SpecPIM to accelerate speculative inference on the PIM-enabled system. In the following sections, we will first introduce the architecture and dataflow design space of SpecPIM. Then, we will introduce SpecPIM's DSE framework, which can find the optimal SpecPIM configurations under different scenarios.

4 SpecPIM Architecture

4.1 Architecture Overview

SpecPIM employs multiple devices for speculative inference. As illustrated in Figure 6-(a), all devices are connected with homogeneous interconnects (e.g., NVLink, PCIe, or CXL) and share a common communication bandwidth. Since different models may have distinct architecture preferences, we assign unique devices to each model. For example, in Figure 6-(a), there are three DLMs and one TLM, which are allocated 1/1/2/4 devices, respectively. The devices for the same model are identical, while the devices across different models may feature varying architectural designs.

The architecture overview of each device is presented in Figure 6-(b). Each device comprises a centralized host processor, a router, and six PIM chips. The host processor resembles a GPU/TPU core, which is equipped with centralized computation engines such as matrix processing units or tensor

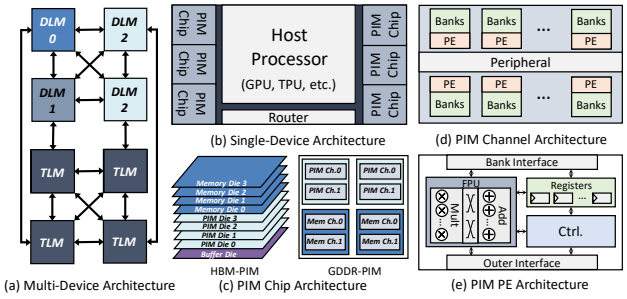

Figure 6. SpecPIM Architecture Overview.

cores for computation-intensive operators. Additionally, the host processor handles the scheduling of the entire inference process based on the dataflow described in Section 5. The router is responsible for inter-device data transfer.

The PIM-chip design is based on existing commodity HBM/GDDR-PIM architectures [31, 33]. As shown in Figure 6-(c), the HBM-PIM chip consists of eight dies stacked on a buffer die. Each die can be either a memory die (1GB/die) or a PIM die (0.5GB/die). In the PIM die, all channels are equipped with PIM PEs. On the GDDR-PIM chip, four packages are present, with each containing two channels. Similar to HBM-PIM, each package can be either a memory package (2GB/package) or a PIM package (1GB/package), and all channels in the PIM package are furnished with PIM PEs.

The overview of the PIM channel is depicted in Figure 6-(d). PEs are integrated near the memory banks. Specifically, HBM-PIM [31] shares each PE with two banks, so there are 32 PEs in one HBM-PIM die. GDDR-PIM [33] places exclusive PEs near every bank, thus also containing 32 PEs in each GDDR-PIM package. Each PE has direct access to its local banks, but there is no direct inter-PE data path, constrained by the scarce routing resources of DRAM technology [11]. All PEs can execute concurrently, thus providing much higher PIM bandwidth than external memory bandwidth.

Figure 6-(e) depicts the architecture overview of PIM PEs. The bank interface connects to local banks, while the outer interface transfers data between the PE and the host processor. The controller drives the floating-point unit (FPU) to conduct computation. The FPU contains multipliers and adders, which can conduct MAC/SIMD operations. The registers are used to buffer intermediate results.

4.2 Architecture Design Space

As listed in Table 2, in SpecPIM's architecture design space, we first explore the memory (PIM) type equipped on PIM chips. Existing commodity PIM designs [31, 33] report different specifications, which can bring disparate effects on the performance. For example, compared with GDDR memory, HBM has higher external bandwidth (307GB/s per HBM cube [36] vs. 256GB/s per four GDDR packages [34], both with 8GB memory) and lower memory access energy (5.47pJ/b vs. 6.48pJ/b [25]). When host processors meet memory-bound issues as discussed in Section 3.1, HBM can bring better performance. On the other hand, GDDR-PIM reports higher computation capacity (0.6TFLOPS per two HBM-PIM dies vs. 1TFLOPS per GDDR-PIM package, both with 1GB memory) due to its higher PE frequency brought by MAC-tree optimization. Therefore, GDDR-PIM can bring better performance to PIM operators considering operators typically behave as computation-intensive on DRAM-PIMs. Exploring the memory (PIM) type enables SpecPIM to choose the most suitable commodity PIM design for different models by comprehensively considering such performance effects.

Table 2. SpecPIM's Architecture Design Space

Hierarchy	Parameter	Range
Each LLM	Memory (PIM) Type	{HBM2, GDDR6}
	PIM Module Number (per chip)	HBM2: {0, 2, 4, 6, 8} PIM Dies GDDR6: {0, 1, 2, 3, 4} PIM Packages

Besides, we further explore the number of PIM modules (HBM-PIM dies or GDDR-PIM packages) equipped on each PIM chip in SpecPIM's architecture design space. In this way, SpecPIM can explore the most suitable PIM resource allocation strategy to satisfy the disparate computation patterns between the TLM and the DLMs as discussed in Section 3.

5 SpecPIM Dataflow

In this section, we will elaborate on SpecPIM's dataflow design in a top-down manner: We will first propose the speculative pipeline to optimize the multi-model workflow. Then we will describe the single model's operator mapping and tiling strategies. Finally, we will extract all components to construct SpecPIM's dataflow design space. Since the prefill stage can be efficiently handled by host processors and only incurs little overhead [9], we mainly focus on the decoding stage of speculative inference.

5.1 Speculative Pipeline

Speculative inference introduces interdependence between the DLMs and the TLM. As illustrated in Figure 7-(a), in vanilla speculative inference, the TLM requires all DLMs' outputs for verification, and all DLMs also need to wait for the TLM's verification results to generate new tokens. Since we assign disparate devices to each model, such interdependence results in resource under-utilization because either the DLMs or the TLM will be idle during the decoding procedure.

To alleviate this problem, we propose the speculative pipeline. The key insight is that when all draft tokens pass verification, we can directly use them for next iteration's generation. Therefore, we let all DLMs generate tokens *optimistically* instead of waiting for the TLM: All DLMs generate tokens by first using the draft tokens generated in the previous iteration, and the current iteration's DLM decoding overlaps with the previous iteration's TLM verification. After

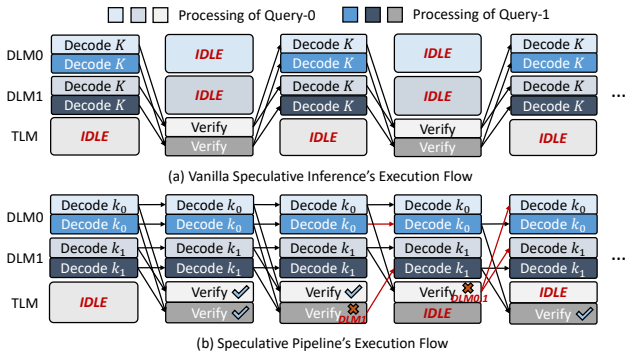

Figure 7. Speculative Inference and Speculative Pipeline.

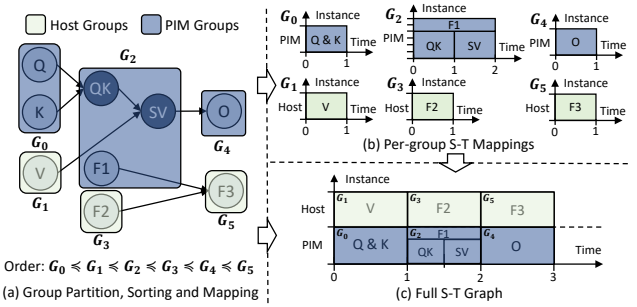

Figure 8. An Example of SpecPIM's Layer Mapping.

the DLMs and the TLM finish execution, speculative inference continues according to TLM's verification results:
Case 1: If all DLMs pass the verification, their generation results for the current iteration are based on correct contexts. Therefore, we can directly send them to the TLM, and the optimistic token generation procedure continues.
Case 2: Otherwise, if only a part of the DLMs passes verification, they will continue optimistic decoding. The wrong DLMs will adopt the revised tokens from the TLM to generate tokens in the next iteration. The TLM will stop verifying in the next iteration so that the wrong DLMs can correct the contexts. More wrong DLMs lead to less opportunity to overlap with the TLM.

Figure 7-(b) illustrates an example of the speculative pipeline. In this example, there are two queries in the batch and we adopt two DLMs for draft token generation, each of which generates k_i (i=0, 1) tokens per iteration. In the second iteration, since all DLMs pass the verification, we can directly use the optimistic outputs for future processing. In the third iteration, the DLM1 does not pass query 1's verification, so it will use TLM's outputs for token generation, and the DLM0 will keep optimistic generation. In the fourth iteration, both DLMs fail to pass query 0's verification. Therefore, the TLM does not need to verify query 0 in the next iteration, and both DLMs will use TLM's outputs for token generation.

5.2 Single Model's Operator Mapping Strategy

For each LLM, we first follow the stage partition strategy adopted by previous works [40, 44, 59, 71] when we use multiple devices: All layers (L layers in total) are evenly partitioned into S stages. Accordingly, all devices (D devices in total) are also evenly partitioned into S parts. Different stages are executed in a pipeline manner, and the original batch (with a batch size of B) is split into M micro-batches (when $S > 1$) to fill the pipeline. For DLMs, we also require $M \geq S$ to avoid pipeline stall caused by the token dependency of autoregressive decoding. In each stage, we execute all layers sequentially, and we adopt the same mapping strategy for all layers since their operator graphs are identical.

For each layer, the inter-operator parallelism can be explored from two levels: (1) Parallelism between the host processor and PIM PEs. (2) Parallelism among different host/PIM instances. To fully explore these parallelism opportunities, we decompose operator mapping into two phases and propose the following formulations to describe the mapping strategy. To simplify the mapping space, we fuse the element-wise operators with their preceding GEMM operators due to their limited amount of mathematical operations.

For a given layer operator graph G, we first partition its operators into K groups and sort them as $G_0 \preceq G_1 \preceq ... \preceq G_{K-1}$, where $G_i \preceq G_j$ indicates that operators in G_i do not depend on operators in G_j. Each operator is assigned to only one group, and these K groups cover all operators in the graph. Subsequently, these operator groups are mapped to one type of accelerator (i.e., the host processor or PIM PEs):

$$AccMap : \{G_0, ..., G_{K-1}\} \rightarrow \{Host, PIM\} \quad (3)$$

Each group takes up all host/PIM resources. Therefore, the groups assigned to the same type of accelerator are executed sequentially, while the groups assigned to different accelerators can be executed in parallel if they have no dependency.

Next, we need to allocate the host/PIM instances to the operators in each group. We first fuse the operators sharing the same input in each group. Assuming group G_i contains P_i operators after fusion, and its corresponding accelerator instance set is $A = \{a_0, ..., a_{N-1}\}$, where a_j represents a host processor or a PIM module (HBM-PIM die or GDDR-PIM package). The intra-group resource allocation can be formulated as the following Space-Time Mapping (S-T Mapping):

$$STMap_{G_i} : G_i \rightarrow (P(A) - \{\varnothing\}) \times [0, P_i - 1] \times [1, P_i] \quad (4)$$

Given an operator $o \in G_i$, $STMap_{G_i}(o)[0]$ indicates the operator's space mapping, and $P(A)$ indicates the power set of A, whose elements are the subsets of accelerator instances. $STMap_{G_i}(o)[1]$ and $STMap_{G_i}(o)[2]$ indicate the intra-group start/end timestamp of each operator, which jointly construct each operator's (logical) time mapping. Each operator's logical duration is 1 by default (i.e., $STMap_{G_i}(o)[2] - STMap_{G_i}(o)[1] = 1$). When an operator is overlapped with multiple operators, its logical duration will be

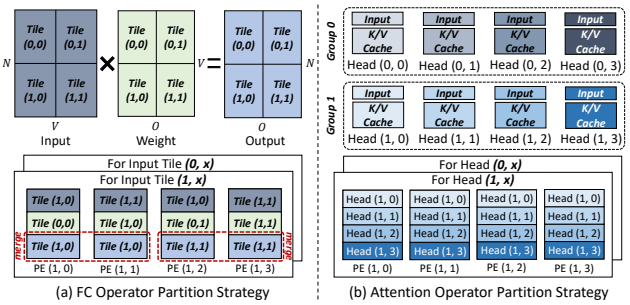

Figure 9. PIM Operator Partition Strategy.

longer than 1. A valid S-T mapping should satisfy the following three constraints: (1) Operator dependency constraint: If operator o_i depends on o_j, o_i's start timestamp should be no earlier than o_j's end timestamp (i.e., $STMap_{G_i}(o_i)[1] \geq STMap_{G_i}(o_j)[2]$). (2) Instance exclusive constraint: At any timestamp, each accelerator instance should be occupied by only one operator. (3) Instance utilization constraint: At any logical timestamp, all accelerator instances should be fully occupied to maximize the hardware utilization under the intra-group operator dependency.

Figure 8 illustrates an example of PaLM's decoder layer [10]. In Figure 8-(a), we split this layer into six groups and sort them as $G_0 \preceq \ldots \preceq G_5$. G_1, G_3, G_5 are mapped to host and G_0, G_2, G_4 are assigned to PIM. Then, in the S-T mappings of Figure 8-(b), the operators in G_0 are first fused together since they share the same inputs. For G_2, we map two PIM instances to operator F1 and the remaining four instances to the other operators. F1 is executed concurrently with the other two operators. For groups with only one operator (post-fusion), all instances are allocated to the sole operator within each group. Finally, we can get the entire space-time graph (S-T graph) of this layer as depicted in Figure 8-(c).

5.3 Single Operator's Tiling Strategy

Considering each operator may be executed on multiple host processors or PIM PEs, we need to conduct tiling to allocate the computation tasks. We first consider the PIM operator's tiling. As described in Section 4.1, since each PE can only directly access its local memory, existing DRAM-PIMs adopt *offloading-based execution model* in three steps: (1) The host scatters inputs to PIM PEs. (2) All PEs conduct computation concurrently. (3) After finishing the computation, the host reads and merges all PEs' results. Given this workflow, we adopt the following tiling strategies for PIM operators:

FC Operator: Since all inputs in one batch share the same weights, we can coalesce them into one matrix multiplication. As illustrated in Figure 9-(a), we can split N, V, and O into $n_N/n_V/n_O$ tiles, respectively. Partitioning the three dimensions brings different trade-offs to PIM operators: (1) Splitting N results in weight duplication, but it reduces the input row vector number processed by each PE, which is more friendly to DRAM-PIMs [36]. (2) Splitting V incurs the overhead of reading and merging partial sums, but it avoids duplicating the inputs to different PEs. (3) Splitting O results in input duplication, but it avoids the partial sum reduction overheads. In the example of Figure 9-(a), there are 8 PEs in total, and n_N=n_V=n_O=2. The two copies of the weight matrix (brought by n_N) are scattered into disparate PE groups. The input matrix is duplicated twice (brought by n_O) and scattered according to the weight tiles' location. The two partial sum tiles (brought by n_V) of the same output tile need to be merged by the host processor.

Attention Operator: Since each K/V head has its own K/V cache and conducts GEMMs disparately, we first split them into n_H groups. The heads in the same group are executed sequentially on the same PEs, while different groups are executed concurrently on distinct PEs. For example, in Figure 9-(b), eight heads are organized into two groups ($n_H = 2$), and the PIM PEs are partitioned into two groups accordingly. Considering all heads have identical shapes, we adopt the same tiling strategy for them, which can be described using the same factors as FC's tiling strategy.

Element-wise Operator: For element-wise operators fused with PIM operators, since each PE may produce partial sums, we execute them in the result merging stage.

For the host operators, we can adopt the same four factors (n_N, n_V, n_O, n_H) to describe the tiling strategies when they are processed by multiple hosts. Different from PIM operators, all groups in attention operators can be executed one-shot by the corresponding host processor.

Table 3. SpecPIM's Dataflow Design Space

Hierarchy	Parameter	Range
Speculative Pipeline	Generation Length (for each DLM)	2 to 5
Each Model	Stage Number S	Common Divisors of $D\&L$
	Micro-barch Number M	Divisors of B
	Group Partition & Sorting	Enumeration
	Group-Accelerator Mapping	Enumeration
	Intra-Group S-T Mappings	Enumeration
Each Operator	Tiling Factors n_N, n_V, n_O, n_H	$n_H \times n_N \times n_V \times n_O = \#PU$

5.4 Dataflow Design Space

SpecPIM's dataflow design space is summarized in Table 3. In the speculative pipeline, we need to explore each DLM's token generation length per iteration. Since the draft tokens need to be accumulated when only a part of DLMs pass verification, we shrink the range of generation length to balance the trade-off discussed in Section 3.1. For each model, the operator group partition & sorting, group-accelerator mapping, and the intra-group S-T mappings can be enumerated given the operator graph and the device architecture. For each operator, the product of four tiling factors should be equal to the processing unit number $\#PU$ (host number or PIM PE number). For FC operators, we always set $n_H = 1$.

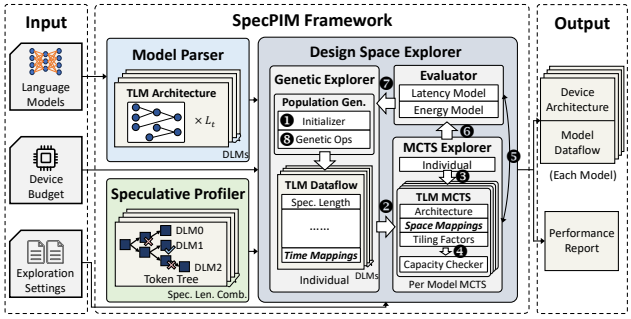

Figure 10. Overview of SpecPIM's DSE Framework.

6 SpecPIM DSE Framework

6.1 Framework Overview

Figure 10 illustrates SpecPIM's DSE framework. Its input contains three components: (1) the TLM and DLMs used by speculative inference, (2) each model's device number budget, and (3) exploration settings (e.g., iteration rounds, optimization goal, etc.). The model parser extracts each model's architecture information (e.g., layer operator graph, layer number). The speculative profiler uses these models to get the statistical information of speculative inference (e.g., the average decoding iteration number, the average token tree length, etc.) under different DLM generation length combinations. This profiling procedure can be conducted offline as long as the TLM and DLMs are determined. After getting all the necessary information, the design space explorer searches for the optimal design iteratively. After finishing DSE, the best design and its performance estimation will be reported. In the next two sub-sections, we will describe the evaluation metrics and the DSE workflow in detail.

6.2 Evaluation Metrics

Based on the dataflow design discussed in Section 5, the design space explorer adopts the following performance models to evaluate the latency and energy of a given design:

Latency Model: We begin with modelling single operator's latency (t_{op}). For the host operators, the total latency is composed of kernel execution latency (t_{kernel}) and the collective communication latency (t_{comm}). For the PIM operators, we also need to consider the latency of input sending (t_{input}) and output fetching (t_{output}) as discussed in Section 5.3:

$$t_{op} = \begin{cases} t^{host}_{kernel} + t_{comm}, & Host\ Ops \\ t_{input} + t^{PIM}_{kernel} + t_{output} + t_{comm}, & PIM\ Ops \end{cases} \quad (5)$$

In Equation (5), $t^{host}_{kernel}, t^{PIM}_{kernel}, t_{comm}$ can be profiled offline like previous works [71]. t_{input}, t_{output} can be estimated according to the data transfer volume, the memory bandwidth, and the PIM instance allocation. After getting each operator's latency, their intra-group finish time can be calculated according to the S-T mapping, and each group's execution latency is the latest finish time of its operators.

Given the group sorting $G_0 \preceq G_1 \preceq ... \preceq G_{K-1}$ and each group's execution latency t_{G_i}, each group's intra-layer finish latency ($t^{fin}_{G_i}$) can be formulated as:

$$t^{fin}_{G_i} = \begin{cases} t_{G_i}, & i = 0 \\ \max\limits_{G_j \in DepSet_{G_i}} \{t^{fin}_{G_j}\} + t_{G_i}, & i > 0 \end{cases} \quad (6)$$

Group G_i's dependent set ($DepSet_{G_i}$) contains: (1) Groups which G_i is dependent on. (2) Groups that are mapped to the same accelerator (host/PIM) as G_i and are sorted before G_i. The layer execution latency is the latest finish time of all groups: $t_{layer} = \max\limits_{0 \le i \le K-1} \{t^{fin}_{G_i}\}$.

Since each stage consists of identical decoder layers, its latency can be estimated as $t_{stage} = L_{stage} \times t_{layer}$, where L_{stage} is the decoder layer number of each stage. Considering the TLM processes tokens in one shot, it launches the pipeline only once for each micro-batch. On the other hand, the DLMs process tokens autoregressively, so they need to launch the pipeline for every token in each micro-batch. Assuming the $i\text{-}th$ DLM generates T_i tokens per iteration, and model X's stage/micro-batch number is S_X/M_X, each model's per iteration latency can be formulated as:

$$\begin{cases} t_{TLM} = t^{TLM}_{stage} \times (S_{TLM} + M_{TLM} - 1) \\ t_{DLM_i} = t^{DLM_i}_{stage} \times (S_{DLM_i} + M_{DLM_i} \times T_i - 1) \end{cases} \quad (7)$$

After getting each model's latency, the speculative inference's latency (t_{spec}) can be formulated as:

$$t_{spec} = (t^{max}_{DLM} + t_{TLM}) + max\{t^{max}_{DLM}, t_{TLM}\} \times (Iter - 1) \quad (8)$$

In Equation (8), the average decoding iteration number $Iter$ can be profiled in the speculative profiler, and t^{max}_{DLM} indicates the longest DLM execution latency.

Energy Model: The total energy (E_{spec}) is the sum of each host kernel's energy (E_{host}), each PIM kernel's energy(E_{PIM}), host-PIM communication energy($E_{host-PIM}$), and the inter-device communication energy(E_{comm}):

$$E_{spec} = \sum E_{host} + \sum E_{PIM} + \sum E_{host-PIM} + \sum E_{comm} \quad (9)$$

6.3 DSE Workflow

The architecture and dataflow of multiple models jointly construct a vast and complex design space. To efficiently explore this space, we propose a DSE algorithm based on the genetic algorithm and the Monte Carlo Tree Search (MCTS) [60]. As illustrated in Figure 10, the algorithm works as follows:

In the beginning, the genetic explorer randomly samples individuals from the design space to construct the initial population (❶). Each individual specifies all models' dataflow, except each group's space mapping and each operator's tiling factors. Then, this population is sent to the MCTS explorer (❷). Each model's partial design in an individual is processed by a distinct MCTS searcher (❸). In each MCTS searcher, the first two layers explore the architectural design and the space-mapping combination for all operator groups, while

the subsequent layers focus on tiling factor exploration. After reaching a leaf node, we can obtain a complete design for this model. If it can pass the capacity checker (❹), it will be sent to the evaluator to get the performance (❺), which is used to update the upper confidence bounds (UCB) bound.

Following the MCTS search, all valid individuals are sent to the evaluator to get the speculative inference performance (❻), with the Top-K individuals fed back to the genetic explorer (❼). For each of the Top-K individuals, we first remove the components searched in MCTS explorer, then mutate each model's dataflow design using one of the three operations to generate new individuals (❽): (1) Randomly change the generation length, the stage number, and the micro-batch size. (2) Randomly change the group-accelerator mapping and the intra-group time mappings. (3) Re-generate operator grouping, group-accelerator mapping, and intra-group time mappings. Throughout these operations, all other components not mentioned remain unchanged. The new population will repeat ❷-❽ in the next iteration. After several iterations, the DSE terminates and the best design will be reported.

The optimization goals of MCTS and Top-K selection can be customized in exploration settings. By default, we take latency as MCTS's optimization goal and energy-delay-product (EDP) as Top-K selection's optimization goal.

Capacity Checker: In current commodity HBM/GDDR-PIM designs, the memory equipped with PIM PEs cannot be accessed by the host processor when PIM PEs are conducting computation [33, 36]. Therefore, to ensure the feasibility of each generated dataflow, we introduce the capacity checker into our DSE framework, which works as follows:

For each model, the capacity checker first assumes all host operators' tensors are placed in normal memory. If the normal memory is overflowed, the capacity checker will adjust the host operators' tensors from normal memory to PIM memory until the overflow issue is solved. To minimize the extra overhead brought by re-allocation, the capacity checker first adjusts host groups that do not overlap with PIM groups. If normal memory is still overflowed after all non-overlapped groups are re-allocated, host groups that are overlapped with PIM groups will be re-allocated, and the logical timestamp & dependent set of operator groups will be updated accordingly. For each type of group (non-overlapped or overlapped), the group with a higher memory footprint will be first reallocated. After finishing the re-allocation, the capacity checker examines whether PIM memory is overflowed. If so, the design will be marked as illegal and discarded.

7 Evaluation

7.1 Evaluation Methodology

Benchmarks: As listed in Table 4, we choose three benchmarks with disparate model configurations to evaluate SpecPIM's performance. All models are under FP16/BF16 data type, and each benchmark is assigned to a different device budget according to its scale. For the models not in original model specs (LLaMA/PaLM 1.3/2.7B), we adopt the same operator graph as their model family and choose the hidden dim & layer number from OPT's model specs [70].

Table 4. Benchmark Configurations

Model Scale	TLM	TLM Device Budget	DLMs	DLM Device Budget
Small	OPT 13B	1	OPT 1.3B	1
Medium	LLaMA 32.5B	2	LLaMA 1.3B/2.7B	(1, 1)
Large	PaLM 62B	4	PaLM 1.3B/2.7B/8B	(1, 1, 2)

Baselines: We compare SpecPIM with four existing GPU/PIM baselines: (1) TLM's autoregressive decoding on GPUs (GPU-AD). (2) Speculative inference on GPUs (GPU-SI). (3) Speculative inference on HBM-PIM-based devices with all FC operators offloaded to PIM (HBM-PIM-FC). (4) Speculative inference on HBM-PIM-based devices with all attention operators offloaded to PIM (HBM-PIM-Attention). The GPU baselines are evaluated on A100 40GB GPUs connected with 600GB/s NVLink and are implemented with DeepSpeed Inference [3]. The devices in HBM-PIM baselines share the same architecture with SpecPIM, except we equip 4 PIM dies to all HBM cubes following Samsung's product [25]. In these baselines, considering speculative inference introduces model dependency, we let the TLM take up all devices and explore the optimal device allocation among the DLMs to fully utilize the hardware resources. The total number of devices is the same as the total device budget in Table 4.

Configuration: We choose NVIDIA A100 GPU as SpecPIM's host processor. For PIM PEs, we adopt Samsung's HBM-PIM design [36] and SK-Hynix's GDDR-PIM design [34]. Each HBM-PIM PE is placed near every two banks and equipped with a 16-lane FP16 FPU @ 300MHz and 16 32B registers. Each GDDR-PIM PE is placed near every bank and equipped with a 16-lane BF16 FPU @ 1GHz, and PEs in one channel share a 2KB global buffer. For fair comparison, we use 5 PIM chips during evaluation, which provides the same maximum capacity as the A100 GPUs we use. We also assume all SpecPIM devices are connected with 600GB/s NVLink.

Profiling: We follow Alpa [71]'s profiling method to get host kernels' execution latency and the collective communication latency. We extend the PIM simulator officially released by Samsung [47] to profile PIM kernels' execution latency. As to the speculative profiler, we use the weights released on Huggingface [22] for profiling. For the models without publicly available weights, we use the weights of OPT models with similar model scales on Huggingface [22] for profiling.

Simulation: We extend Samsung's PIM simulator [47] and inject the profiled host kernel latency into our simulator to support single-device cycle-accurate simulation. We adopt the event-driven method used by AlpaServe [40] to get multi-device performance. For energy evaluation, we adopt PyNVML [53] to measure host kernels' energy. The energy consumption of HBM-PIM and GDDR-PIM PEs are 1.012pJ and 0.870pJ per MAC according to [36, 58]. The memory access

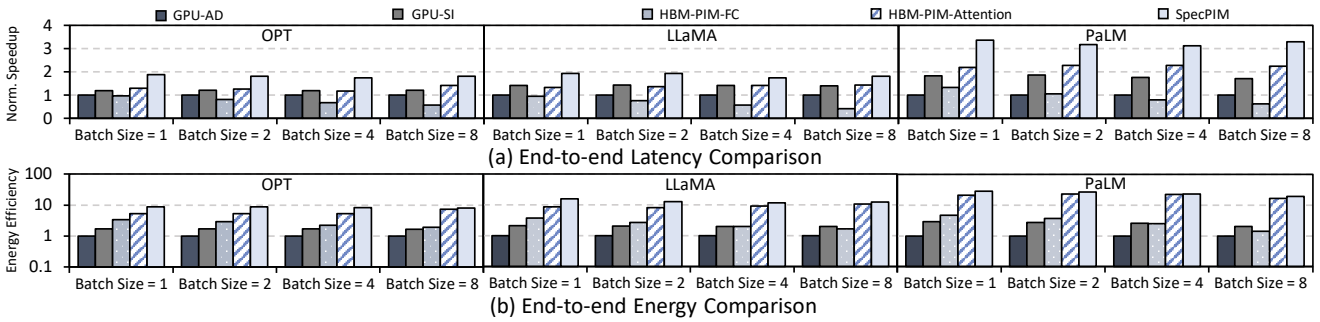

Figure 11. End-to-end Energy Performance Comparison.

energy of HBM2 and GDDR6 are 5.47pJ/b and 6.48pJ/b according to TPU-v4i [25]. NVLink's energy is 1.3pJ/b according to NVIDIA's Blog [46].

7.2 End-to-end Performance

We first compare SpecPIM's end-to-end performance against the baselines. For each benchmark, we set the prompt length, generation length, and KV Cache's context length to 1024, and we evaluate the performance under the batch size of 1/2/4/8. For SpecPIM's DSE, we iterate the genetic algorithm for 50 rounds and sample 1000 individuals per iteration. In each model's MCTS, we search for 10000 leaf nodes. During individual mutation, we select the Top-50 individuals in each population. The latency and energy results are depicted in Figure 11, where all results are normalized to GPU-AD.

Latency: As Figure 11-(a) shows, GPU-SI brings 1.45× geomean speedup compared with GPU-AD. For HBM-PIM-FC, it cannot bring speedup to speculative inference and is even slower than the GPU-AD in some cases. The reasons are twofold: (1) The FC operators in TLMs are more computation intensive, which cannot be efficiently handled by DRAM-PIMs. (2) The limited PIM capacity (10 GB/device) cannot hold all data in FC operators, so we have to place overflowed data to normal memory and move them to PIMs repeatedly, which incurs 38.8% geomean extra overhead. For HBM-PIM-Attention, it offloads fewer operators to PIM and avoids the PIM capacity overflow issue. However, it only brings 1.09× geomean speedup against GPU-SI because DLMs have more PIM-friendly operators and remain unexplored by HBM-PIM-Attention. In contrast, SpecPIM achieves 2.21×/1.52× geomean speedup compared with GPU-AD/GPU-SI. Compared with HBM-PIM-FC/HBM-PIM-Attention, SpecPIM can achieve 2.91×/1.39× geomean speedup (2.02× geomean speedup on all PIM baselines). SpecPIM brings much higher speedup to speculative inference than existing solutions owing to the exploration of the heterogeneity-aware design space.

Energy Efficiency: In Figure 11-(b), we can find that GPU-SI brings 2.07× geomean better energy efficiency than GPU-AD. HBM-PIM-FC only brings 1.25× better geomean energy efficiency against GPU-SI, and it reports poorer energy efficiency than GPU-SI under large batch sizes. Although HBM-PIM-Attention reports 4.95× geomean better energy efficiency against GPU-SI, SpecPIM can bring 6.67× better geomean energy efficiency against GPU-SI owing to the heterogeneity-aware exploration. Compared with HBM-PIM-FC/HBM-PIM-Attention, SpecPIM reports 5.34×/1.35× better geomean energy efficiency (2.68× better geomean energy efficiency on all PIM baselines).

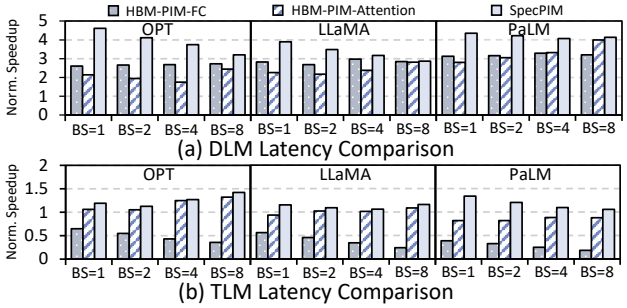

Figure 12. Single Model's Performance Analysis.

7.3 Performance Analysis

To analyze the improvement from the components of SpecPIM's design space, we conduct the following experiments:

Single-model Exploration Analysis: To analyze the speedup brought by SpecPIM's single-model dataflow exploration, we compare with FC-only offloading and Attention-only offloading strategies. We use HBM-PIM-based devices and equip four PIM dies with each PIM chip. We evaluate 1.3B/2.7B/8B DLMs on 1/1/2 devices and 13B/32.5B/62B TLMs on 2/4/8 devices. We let DLMs conduct autoregressive decoding and set the TLMs' verification length to 4. The other settings are kept the same as Section 7.2.

Figure 12 summarizes the evaluation results. All results are normalized to the performance on the same number of GPUs, and we report the geomean performance when there are multiple DLMs. In Figure 12-(a), we can find that all offloading strategies can outperform the GPU baseline on DLMs because GEMMs in DLMs are more friendly to DRAM-PIMs. SpecPIM's dataflow exploration can further optimize

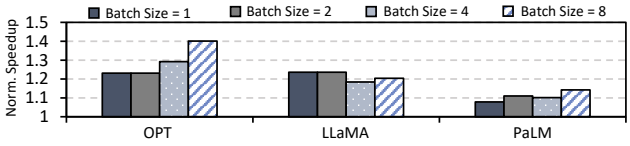

Figure 13. Speculative Pipeline's Performance Analysis.

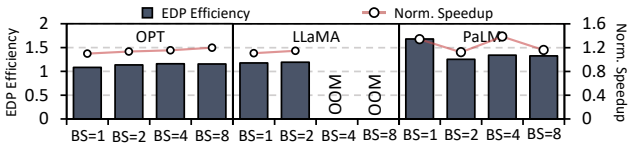

Figure 14. Architecture DSE's Performance Analysis.

the offloading & tiling strategies, thus achieving 1.29×/1.41× geomean speedup against the two baselines.

Figure 12-(b) depicts TLMs' results. We can find that HBM-PIM-FC reports much slower performance than GPU since TLM's FC operators are more computation-intensive. For HBM-PIM-Attention, it can achieve speedup on OPT & LLaMA but behaves poorer than GPU on PaLM. That is because PaLM adopts multi-query attention [56] and shares one KV head with all query vectors, which increases attention operators' arithmetic intensity. In contrast to the two baselines, SpecPIM can not only optimize operator offloading for different models but also explore the inter-operator parallelism to fully utilize computation resources, thus achieving 1.18× geomean speedup against the GPU baseline.

Speculative Pipeline Analysis: To demonstrate the performance improvement brought by speculative pipeline, we compare the performance of SpecPIM's design in Section 7.2 with/without speculative pipeline and depict the comparison results in Figure 13. We can find that using speculative pipeline can bring 1.20× geomean speedup. Besides, speculative pipeline can achieve better performance improvement when there are fewer DLMs because the optimistic decoding has a higher likelihood of continuing.

Architecture Exploration Analysis: To analyze the effect of exploring PIM settings, we compare SpecPIM's performance on a specific device with the results of full exploration. Specifically, we explore SpecPIM's dataflow design on HBM-PIM-based devices with each PIM chip equipping two PIM dies. We adopt the same device budget as Table 4 and keep all other settings the same as Section 7.2. As illustrated in Figure 14, we can find that SpecPIM with architecture exploration brings 1.18× geomean speedup and 1.24× geomean EDP efficiency improvement. Besides, using one type of device encounters out-of-memory (OOM) issues on LLaMA under large batch sizes with the given device budget. Architecture exploration can better meet the disparate requirements under different scenarios.

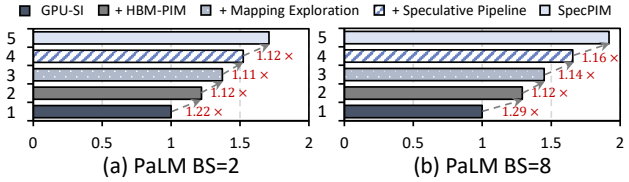

Figure 15. SpecPIM Speedup Breakdown.

Speedup Breakdown: To better illustrate the performance improvement brought by different components in SpecPIM's design space, we showcase the speedup breakdown of speculative inference on PaLM models under small & large batch sizes (2 and 8). We offload the attention operators to PIM before conducting dataflow exploration and adopt HBM-PIM-based devices (4 PIM dies on each HBM-PIM cube) before conducting architecture exploration. As shown in Figure 15, we can find that offloading attention operator to PIM can bring 1.22×/1.29× speedup. By further exploring each model's dataflow design space, we can gain 1.12×/1.12× speedup compared with the fixed offloading strategy. Then, adopting the speculative pipeline can bring 1.12×/1.14× speedup thanks to the overlap between the TLM and the DLMs. Finally, if we can explore the architecture design space together with the dataflow design space, we can further gain 1.12×/1.16× speedup. Generally, our designs can effectively accelerate speculative inference on the PIM-enabled system.

7.4 Design Space Visualization

In Figure 16, we visualize SpecPIM's design space under small & large batch sizes (2 and 8) using the populations sampled in Section 7.2. Apart from SpecPIM's design space, we also plot the following baselines' relative performance: (1) Speculative inference on GPUs (GPU-SI). (2) Speculative inference on Samsung HBM-PIM [36] based devices (i.e., four PIM dies per cube) with FC offloading (HBM-PIM-FC) and Attention offloading (HBM-PIM-Attention) strategies. (3) Speculative inference on SK-Hynix GDDR-PIM [34] based devices (i.e., all packages equipped with PIM) with FC offloading (GDDR-PIM-FC) and Attention offloading (GDDR-PIM-Attention) strategies. All baselines adopt the same baseline dataflow described in Section 7.1. The out-of-range normalized energy consumption is notated near the corresponding points.

Architectural Implications: From the DSE results, we can first draw some architectural implications. In Figure 16, we can find that GDDR-PIM-FC can always outperform HBM-PIM-FC when the out-of-memory (OOM) issue is not encountered. The reason is two-fold: First, considering the FC-offloading strategy executes most operators on DRAM-PIMs, GDDR-PIM's higher computation capacity can bring better performance to the PIM operators, which is more friendly to such a PIM-centric execution strategy.

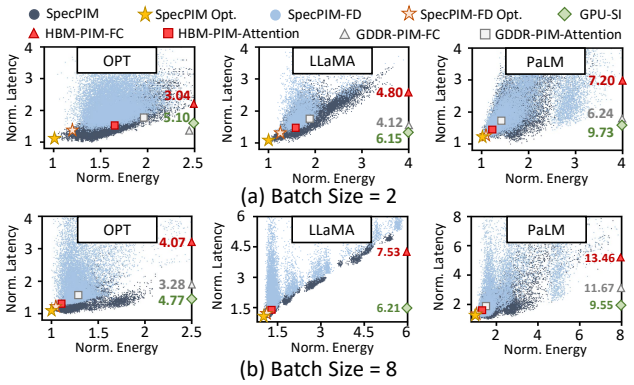

Figure 16. Design Space Visualization (Omit OOM Points).

Takeaway 1: Future PIM design should enhance the computation capacity to boost the performance of PIM operators.

Second, HBM-PIM's PIM memory capacity cannot hold all FC operators' tensors (especially for TLMs), which incurs extra overhead to transfer data between the normal memory and the PIM memory. Therefore, the operator mapping strategy should be aware of the underlying architecture to better utilize all hardware components.

Takeaway 2: The operator offloading strategy should be flexible to fit in with the underlying architecture.

On the other hand, GDDR-PIM-Attention behaves poorer than HBM-PIM-Attention. The main reason is that the Attention-based offloading strategy executes most operators on the host processors. The higher memory bandwidth and lower memory access energy of HBM memory are more friendly to the host operators, thus bringing better performance to HBM-PIM-Attention.

Takeaway 3: Future PIM design should also improve the external memory bandwidth to boost the performance of host operators.

Compared with these baselines, SpecPIM can always find more efficient designs owing to its extensive exploration in the heterogeneity-aware design space. In these benchmarks, we also find that SpecPIM typically assigns fewer PIM resources ($\leq$ 50% of PIM die/packages) to the TLMs and allocates more PIM resources to the DLMs ($\geq$ 50% of PIM die/packages), which is also proportional to the ratio of PIM operators. Such a tendency corresponds to the computation properties of TLMs and DLMs as discussed in Section 3.1.

Takeaway 4: Assigning more PIM resources to DLMs and fewer PIM resources to TLMs can better satisfy their disparate computation patterns in speculative inference.

DSE under Real-world Products: Considering it might be difficult to get all types of devices listed in Table 2, we also conduct DSE under a fixed set of devices based on the real-world products: (1) HBM-based device without PIM dies (A100 GPU). (2) HBM-based device with four PIM dies per HBM cube (Samsung HBM-PIM). (3) GDDR-based device with all packages equipped with PIM (SK-Hynix GDDR-PIM). We use the same DSE settings as Section 7.2 and plot the design space in Figure 16 (denoted as SpecPIM-FD). Although exploring on fixed device set reports 1.22×/1.02× geomean higher latency/energy consumption compared with SpecPIM's full exploration as to the optimal performance, it can still find out design better than all baselines. Compared with the best baselines, SpecPIM-FD achieves 1.12×/1.26× geomean speedup/energy efficiency. Therefore, heterogeneity-aware exploration is also meaningful to accelerate speculative inference under real-world cases.

8 Conclusion

This paper proposes SpecPIM, which takes the first trial to accelerate speculative inference on the PIM-enabled system via architecture-dataflow co-exploration. SpecPIM constructs the architecture and dataflow design space by comprehensively considering the algorithmic heterogeneity in speculative inference and the architectural heterogeneity between centralized computing and DRAM-PIMs. Based on the co-design space, SpecPIM conducts DSE to find out the optimal design under different scenarios. Compared with speculative inference on GPUs and existing PIM-based LLM accelerators, SpecPIM achieves 1.52×/2.02× geomean speedup and 6.67×/2.68× geomean higher energy efficiency.

Acknowledgments

We thank all the reviewers and our shepherd for their valuable comments. This work is supported by National Natural Science Foundation of China (NSFC) (Grant No. 62032001) and 111 Project (B18001). This work is also supported in part by the NSFC under Grant No. U21B2017.

References

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 105–117, 2015.

[2] Amazon. Bedrock. https://aws.amazon.com/bedrock/.

[3] 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. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[4] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Sung-Kyu Lim, Hyesoon Kim, et al. Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 908–920. IEEE, 2021.

[5] 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 *2016 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam,

Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[7] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[8] 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. *arXiv preprint arXiv:2107.03374*, 2021.

[9] 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*, 2023.

[10] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[11] Fabrice Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24. IEEE Computer Society, 2019.

[12] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 283–295. IEEE, 2015.

[13] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 113–124. IEEE, 2015.

[14] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–764, 2017.

[15] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 807–820, 2019.

[16] Github. Copilot. https://github.com/features/copilot.

[17] Google. Bard. https://bard.google.com/.

[18] Peng Gu, Xinfeng Xie, Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, and Yuan Xie. ipim: Programmable in-memory image processing accelerator using near-bank architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 804–817. IEEE, 2020.

[19] Byungchul Hong, Yeonju Ro, and John Kim. Multi-dimensional parallel training of winograd layer on memory-centric architecture. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 682–695. IEEE, 2018.

[20] Seongmin Hong, Seungjae Moon, Junsoo Kim, Sungjae Lee, Minsub Kim, Dongsoo Lee, and Joo-Young Kim. Dfx: A low-latency multi-fpga appliance for accelerating transformer-based text generation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 616–630. IEEE, 2022.

[21] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. Transparent offloading and mapping (tom) enabling programmer-transparent near-data processing in gpu systems. *ACM SIGARCH Computer Architecture News*, 44(3):204–216, 2016.

[22] Huggingface. Models - huggingface,. https://huggingface.co/models.

[23] Liancheng Jia, Zizhang Luo, Liqiang Lu, and Yun Liang. Tensorlib: A spatial accelerator generation framework for tensor algebra. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 865–870. IEEE, 2021.

[24] Joao Gante. Assisted generation: a new direction toward low-latency text generation, 2023.

[25] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter C. Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David A. Patterson. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2021.

[26] Sheng-Chun Kao and Tushar Krishna. Magma: An optimization framework for mapping multiple dnns on multiple accelerator cores. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 814–830. IEEE, 2022.

[27] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S Lee, et al. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 790–803. IEEE, 2020.

[28] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shinhaeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, Kyung-Soo Kim, Jin Jung, IlKwon Yun, Sung Joo Park, Hyunsun Park, Joon-Ho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin S. Lee. Near-memory processing in action: Accelerating personalized recommendation with axdimm. *IEEE Micro*, 42(1):116–127, 2021.

[29] Duckhwan Kim, Taesik Na, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Deeptrain: A programmable embedded platform for training deep neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2360–2370, 2018.

[30] Jin Hyun Kim, Shinhaeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin,

BengSeng Phuah, Jihyun Choi, Jinin So, YeonGon Cho, Joon-Ho Song, Jangseok Choi, Jeonghyeon Cho, Kyomin Sohn, Young-Soo Sohn, Kwang-Il Park, and Nam Sung Kim. Aquabolt-xl: Samsung hbm2-pim with in-memory processing for ml accelerators and beyond. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–26. IEEE, 2021.

[31] Jin Hyun Kim, Yuhwan Ro, Jinin So, Sukhan Lee, Shinhaeng Kang, YeonGon Cho, Hyeonsu Kim, Byeongho Kim, Kyungsoo Kim, Sangsoo Park, Jin-Seong Kim, Sanghoon Cha, Won-Jo Lee, Jin Jung, Jong-Geon Lee, Jieun Lee, Joon-Ho Song, Seungwon Lee, Jeonghyeon Cho, Jaehoon Yu, and Kyomin Sohn. Samsung pim/pnm for transfmer based ai: Energy efficiency on pim/pnm cluster. In *2023 IEEE Hot Chips 35 Symposium (HCS)*, pages 1–31. IEEE Computer Society, 2023.

[32] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE micro*, 40(3):20–29, 2020.

[33] Yongkee Kwon, Guhyun Kim, Nahsung Kim, Woojae Shin, Jongsoon Won, Hyunha Joo, Haerang Choi, Byeongju An, Gyeongcheol Shin, Dayeon Yun, Jeongbin Kim, Changhyun Kim, Ilkon Kim, Jaehan Park, Chanwook Park, Yosub Song, Byeongsu Yang, Hyeongdeok Lee, Seungyeong Park, Wonjun Lee, Seongju Lee, Kyuyoung Kim, Daehan Kwon, Chunseok Jeong, John Kim, Euicheol Lim, and Junhyun Chun. Memory-centric computing with sk hynix's domain-specific memory. In *2023 IEEE Hot Chips 35 Symposium (HCS)*, pages 1–26. IEEE Computer Society, 2023.

[34] Yongkee Kwon, Kornijcuk Vladimir, Nahsung Kim, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Guhyun Kim, Byeongju An, Jeongbin Kim, Jaewook Lee, Ilkon Kim, Jaehan Park, Chanwook Park, Yosub Song, Byeongsu Yang, Hyungdeok Lee, Seho Kim, Daehan Kwon, Seong Ju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeong Pil Kang, Jungyeon Kim, Junyeol Jeon, Myeongjun Lee, Minyoung Shin, Minhwan Shin, Jaekyung Cha, Changson Jung, Kijoon Chang, Chunseok Jeong, Euicheol Lim, Il Park, and Junhyun Chun. System architecture and software stack for gddr6-aim. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–25. IEEE, 2022.

[35] Youngeun Kwon, Yunjae Lee, and Minsoo 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*, pages 740–753, 2019.

[36] Suk Han Lee, Shinhaeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, Seongil O, 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 *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 43–56. IEEE, 2021.

[37] Young Sik Lee and Tae Hee Han. Task parallelism-aware deep neural network scheduling on multiple hybrid memory cube-based processing-in-memory. *IEEE Access*, 9:68561–68572, 2021.

[38] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[39] Cong Li, Zhe Zhou, Xingchen Li, Guangyu Sun, and Dimin Niu. Nm-explorer: An efficient exploration framework for dimm-based near-memory tensor reduction. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[40] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. *arXiv preprint arXiv:2302.11665*, 2023.

[41] Liu Liu, Jilan Lin, Zheng Qu, Yufei Ding, and Yuan Xie. Enmc: Extreme near-memory classification via approximate screening. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1309–1322, 2021.

[42] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. Tenet: A framework for modeling tensor dataflow based on relation-centric notation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 720–733. IEEE, 2021.

[43] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 2023.

[44] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[45] Dimin Niu, Shuangchen Li, Yuhao Wang, Wei Han, Zhe Zhang, Yijin Guan, Tianchan Guan, Fei Sun, Fei Xue, Lide Duan, Yuanwei Fang, Hongzhong Zheng, Xiping Jiang, Song Wang, Fengguo Zuo, Yubing Wang, Bing Yu, Qiwei Ren, and Yuan Xie. 184qps/w 64mb/mm 2 3d logic-to-dram hybrid bonding with process-near-memory engine for recommendation system. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 1–3. IEEE, 2022.

[46] NVIDIA. What is nvlink? https://blogs.nvidia.com/blog/what-is-nvidia-nvlink/.

[47] Samsung Advanced Institute of Technology. Pimsimulator. https://github.com/SAITPublic/PIMSimulator.

[48] OpenAI. Chatgpt. https://openai.com/blog/chatgpt.

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

[50] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 304–315. IEEE, 2019.

[51] 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-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 268–281, 2021.

[52] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[53] Rjzamora. pynvml. https://pypi.org/project/pynvml.

[54] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

[55] Fabian Schuiki, Michael Schaffner, Frank K Gürkaynak, and Luca Benini. A scalable near-memory architecture for training deep neural networks on large in-memory datasets. *IEEE Transactions on Computers*, 68(4):484–497, 2018.

[56] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

[57] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[58] Hyunsung Shin, Dongyoung Kim, Eunhyeok Park, Sungho Park, Yongsik Park, and Sungjoo Yoo. Mcdram: Low latency and energy-efficient matrix computations in dram. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2613–2622, 2018.

[59] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[60] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[61] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Kathleen S. Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Agüera y Arcas, Claire Cui, Marian Croak, Ed H. Chi, and Quoc Le. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.

[62] 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. *arXiv preprint arXiv:2302.13971*, 2023.

[63] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[65] Yi Wang, Weixuan Chen, Jing Yang, and Tao Li. Towards memory-efficient allocation of cnns on processing-in-memory architecture. *IEEE Transactions on Parallel and Distributed Systems*, 29(6):1428–1441, 2018.

[66] 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 *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 570–583. IEEE, 2021.

[67] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 369–383, 2020.

[68] Shouyi Yin, Shibin Tang, Xinhan Lin, Peng Ouyang, Fengbin Tu, Jishen Zhao, Cong Xu, Shuangcheng Li, Yuan Xie, ShaoJun Wei, et al. Parana: A parallel neural architecture considering thermal problem of 3d stacked memory. *IEEE Transactions on Parallel and Distributed Systems*, 30(1):146–160, 2018.

[69] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 85–98, 2014.

[70] 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. *arXiv preprint arXiv:2205.01068*, 2022.

[71] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[72] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 874–887, 2022.

[73] Size Zheng, Siyuan Chen, Siyuan Gao, Liancheng Jia, Guangyu Sun, Runsheng Wang, and Yun Liang. Tileflow: A framework for modeling fusion dataflow via tree-based analysis. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1271–1288, 2023.

[74] Size Zheng, Siyuan Chen, Peidi Song, Renze Chen, Xiuhong Li, Shengen Yan, Dahua Lin, Jingwen Leng, and Yun Liang. Chimera: An analytical optimizing framework for effective compute-intensive operators fusion. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1113–1126. IEEE, 2023.

[75] Zhe Zhou, Cong Li, Xuechao Wei, Xiaoyang Wang, and Guangyu Sun. Gnnear: Accelerating full-batch training of graph neural networks with near-memory processing. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 54–68, 2022.

[76] Zhe Zhou, Cong Li, Fan Yang, and Guangyu Sun. Dimm-link: Enabling efficient inter-dimm communication for near-memory processing. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 302–316. IEEE, 2023.