PRIMATE: Processing in Memory Acceleration for Dynamic Token-pruning Transformers

Yue Pan[1], Minxuan Zhou[1], Chonghan Lee[2], Zheyu Li[2], Rishika Kushwah[2], Vijaykrishnan Narayanan[2], Tajana Rosing[1]
[1]*University of California San Diego*, La Jolla, CA, United States
[2]*Pennsylvania State University*, State College, PA, United States
{yup014, miz087, tajana}@ucsd.edu
{cvl5361, zil5126, rqk5510, vijay}@psu.edu

2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC) | 979-8-3503-9354-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/ASP-DAC58780.2024.10473968

***Abstract*—Attention-based models such as Transformers represent the state of the art for various machine learning (ML) tasks. Their superior performance is often overshadowed by the substantial memory requirements and low data reuse opportunities. Processing in Memory (PIM) is a promising solution to accelerate Transformer models due to its massive parallelism, low data movement costs, and high memory bandwidth utilization. Existing PIM accelerators lack the support for algorithmic optimizations like dynamic token pruning that can significantly improve the efficiency of Transformers. We identify two challenges to enabling dynamic token pruning on PIM-based architectures: the lack of an in-memory top-k token selection mechanism and the memory underutilization problem from pruning. To address these challenges, we propose PRIMATE, a software-hardware co-design PIM framework based on High Bandwidth Memory (HBM). We initiate minor hardware modifications to conventional HBM to enable Transformer model computation and top-k selection. For software, we introduce a pipelined mapping scheme and an optimization framework for maximum throughput and efficiency. PRIMATE achieves 30.6× improvement in throughput, 29.5× improvement in space efficiency, and 4.3× better energy efficiency compared to the current state-of-the-art PIM accelerator for Transformers.**

I. Introduction

Attention-based models like Transformers have gained tremendous ground in many applications in recent years. Large Language Models (LLMs) like ChatGPT have shown astonishing capabilities in understanding and responding to human prompts, but with enormous requirements in computing and memory. Researchers have been exploring solutions from the realms of algorithms and architectures to mitigate LLMs' high costs.

Dynamic token pruning algorithms have been introduced to reduce the computation requirements of Transformers [12], [22]. Taking advantage of redundancy in human language and images, unimportant tokens can be progressively pruned based on their importance. Figure 1 and 2 visualize the dynamic token pruning process. This technique achieves a speedup of 2.3× to 2.7× with 2.6× to 3.2× in GPU memory reduction [12]. Importantly, token pruning often results in comparable or higher accuracy across a wide range of NLP and vision tasks, as can be seen from the Pareto frontier in Figure 3.

Both ASIC and Processing in Memory (PIM) solutions have been extensively explored as hardware accelerators for Transformers. By integrating compute units directly into memory, PIM significantly reduces the data movement cost during computation, which dominates latency and energy in traditional architectures. PIM also exploits the massive parallelism and high internal bandwidth of memory to support high-throughput processing. Previous works [19], [27] have shown a software-hardware co-designed PIM architecture can provide better throughput and power consumption than GPU and TPU for Transformers. However, they lack the support for algorithmic optimizations like dynamic token pruning. One Resistive RAM (ReRAM)

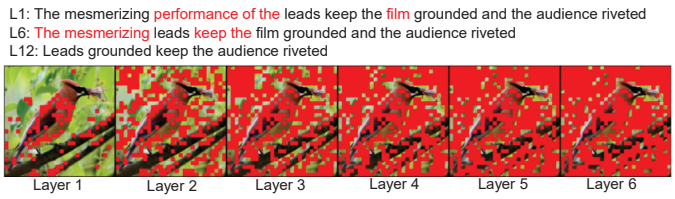

Fig. 1: Dynamic token pruning on text data (SST-2 dataset) and an image (CUB 2011 Bird). Redundant keywords/patches (in red) are progressively pruned during the inference.

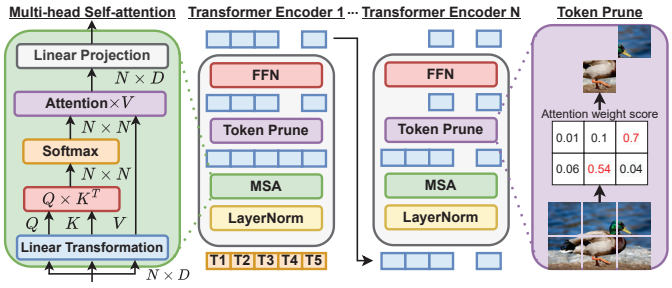

Fig. 2: Overview of Dynamic Token-pruning Transformer on vision tasks.

based architecture was proposed with token pruning support [24], but it suffers from numerical instabilities of ReRAM and requires on-chip re-computation of attention scores, complicating dataflow and introducing additional overhead.

This work focuses on DRAM-based (HBM) PIM technologies which can support larger capacity than SRAM [23] with high bandwidth, lower latency, and higher numerical stability than non-volatile memory designs [6], [24]. However, current designs like [27] face two challenges in enabling token pruning: the lack of top-k selection mechanisms and the memory underutilization problem from pruning. In this work, we address these challenges and propose PRIMATE, a novel in-memory hardware-software co-design acceleration framework that synergizes PIM and Dynamic Token-pruning Transformer. With PRIMATE, we achieve up to 30.6× improvement in throughput, 29.5× better space efficiency, and 4.3× better energy efficiency compared to the current state-of-the-art [27] on popular Transformer models and workloads.

II. Background and Related Work

A. Transformer and Dynamic Token Pruning

The token pruning strategy is to prune tokens based on an *importance score* of each token. To arrive at this score, we take the dot

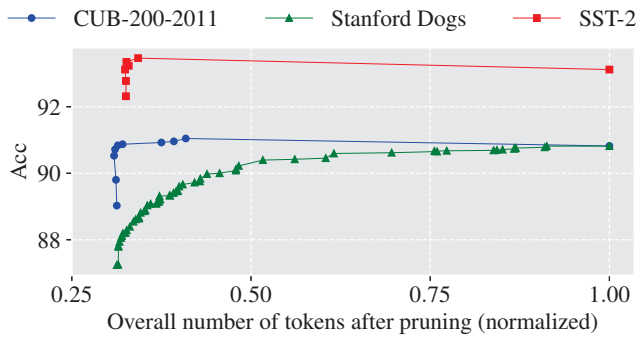

Fig. 3: Model accuracy v.s. remaining tokens

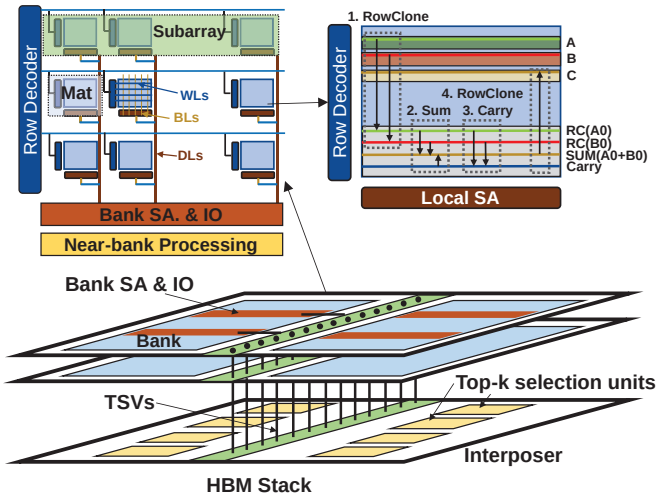

Fig. 4: The HBM-based architecture with PIM and NBP.

product between Q and K vectors to get the attention probabilities. Then, we normalize them using a softmax function and use the normalized attention probabilities to compute a weighted sum of the V vector as shown in Figure 2. During dynamic token pruning, we use the attention probabilities as *importance scores* to keep the top-k most important tokens after each encoder layer. Different from other approaches [16], [25] that prune tokens at the end of each encoder layer, our approach prunes tokens before the feedforward layer within each encoder layer, achieving additional reductions in compute and memory costs. During training, we introduce arbitrary token pruning ratios to enable the model to adapt to various token drop scenarios. We also employ a multi-objective evolutionary search on the trained model to determine layer-wise pruning ratios that achieve the optimal accuracy-efficiency trade-offs. After these efforts, token-pruning models offer comparable or even higher accuracy than the original model with significantly less computing requirements, as seen from the Pareto frontier in Figure 3.

B. Processing In-Memory

Figure 4 shows the architecture of a High Bandwidth Memory (HBM), a high-end commodity DRAM product. The basic unit of memory is a *bank*, which consists of 2D cell arrays and peripherals to transfer data between DRAM cells and IOs. The memory cells are grouped into a set of *subarrays*, each of which consists of a row of mats. Each mat has local sense amplifiers (row buffers) sensing a horizontal wordline (WL) via a set of vertical bitlines (BLs). Sense amplifiers in mats of a subarray form the subarray row buffer. Upon receiving a DRAM access, the bank activates the corresponding WL in the subarray row buffer and transfers the whole WL to the bank-level sense amplifiers via the data lines (DLs).

There have been several DRAM-based PIM technologies that support operations in different levels of DRAM architecture including near-bank processing (NBP) [11], near-subarray processing (NSP) [13], and processing using-memory (PuM) [3]. Many previous works have utilized different PIM technologies to accelerate machine learning workloads [5], [13], [27]. Specifically, Newton [5] proposes near-bank multipliers and adder trees in HBM to accelerate various ML operators. DRISA [13] integrates near-subarray 1-bit logic (e.g., full adder) that exploits subarray-level parallelism. However, when accelerating Transformers, Newton's bank-level processing has limited throughput, while DRISA's near-subarray processing cannot efficiently process long-vector reduction (accumulation). TransPIM [27] proposed a hybrid method that exploits the subarray-level computation for element-wise arithmetic and the near-bank logic for efficient accumulation to accelerate Transformers. TransPIM [27] uses bit-serial PuM technology [3] that only introduces negligible area overhead in commercial HBM while providing high throughput.

Figure 4 illustrates the HBM architecture that supports bit-serial PuM and NBP. Specifically, PuM directly generates the result of computation between different WLs by exploiting the charge-sharing effect of the DRAM mechanism [17]. Figure 4 also shows an example of a 1-bit full-addition that adds one row (bit) of A and B to generate 1 row (bit) of C. We need row clone (RC) operations to backup operands and write back the result. The sum and carry generation is implemented using Activate-Activate-Precharge (AAP) operations [17]. An n-bit multiplication using the bit-serial PIM requires around $7n^2$ AAP operations. Bit-serial PIM can parallelize the computation for whole DRAM rows (e.g., 8K) and exploit the subarray-level parallelism [9] to provide extensive throughput. Due to the bit-serial layout, the NBP unit also works in a bit-serial manner that fully exploits the internal data links.

C. Token-based Data Flow

Like TransPIM [27], we adopt token-based dataflow. This approach maps data and computation to memory based on the corresponding input token at the bank granularity. For each Transformer encoder/decoder layer, token-based flow first distributes the input sequence to all banks, where each bank computes the Q, K, and V vectors with multiple heads for its allocated tokens and the pre-allocated fully connected weights. The self-attention layer consists of two steps. Each bank first computes attention scores between its allocated tokens using computed Q and K values. Then, each bank broadcasts its K values to all other banks to compute global attention scores. We also adopt the lightweight inter-bank connection of previous work [27] to accelerate data broadcast. We compute the attention scores for different heads, each applying a softmax operation. We multiply the multi-head attention score matrix with multi-head V vectors using a similar broadcast phase to calculate the attention output matrix following the token-based layout. Then, the feed-forward layers are processed similarly to the fully connected layer. To enable the token-based data flow, each bank has a duplication of Transformer weights including fully-connected operations

and feed-forward operations. All other operations use the intermediate data corresponding to tokens that can stay in the memory during the whole process.

III. PRIMATE Overview

A. Limitations of Current PIM Acceleration

The existing PIM-based accelerator and token-based data flow cannot efficiently support dynamic token pruning. First, to decide which tokens to prune, we need a top-k selection mechanism based on the token importance score. The current SOTA PIM Transformer accelerator [27] requires frequent intra-memory data movements for attention score accumulation and relies on the host CPU to perform top-k selection. Per our experiment with SOTA [27], relying on the CPU for top-k selection incurs up to 25% penalty of the overall latency. Second, before pruning, the memory is configured with a data layout efficient for full-token computation under the token-based dataflow. However, the remaining tokens after pruning are scattered across the memory, causing spatial under-utilization and larger communication overhead. Due to the delicate data layout requirement of PIM, it is hard to reclaim the memory occupied by pruned tokens, leading to less *effective* memory savings. One remedy is to aggregate the remaining tokens after every layer. However, our experiments in section VI-D show unsatisfactory throughput with this method due to the sub-optimal token-to-bank mappings (details in Section VI-A). For these reasons, existing PIM accelerators for Transformers [27] can only provide 1.5× speedup when the workload allows 3.2× theoretical speedup (69% computation pruned). Therefore, the inefficiencies of existing PIM accelerators at processing Dynamic Token-pruning Transformer necessitates a novel PIM-based architecture that is efficient at supporting the token pruning mechanism.

B. The PRIMATE SW-HW Co-design Framework

To tackle the aforementioned challenges, we propose PRIMATE. First, we design and implement Top-k Engines (TEs), a near-channel accumulation and token selection mechanism that introduce minimal area and power overhead to the HBM. Second, we propose a pipeline scheduling and mapping method to mitigate the under-utilization caused by pruning and leverage inter-layer parallelism when processing a stream of input sequences. Given a pruning configuration, we partition memory into adequately sized blocks for specific layers to enable high spatial utilization throughout inference. Additionally, we propose a framework that progressively optimizes the pipeline mapping scheme for given configurations of memory, model, and token pruning for overall throughput and efficiency. These optimizations are translated into routines of memory instructions and are deployed at runtime.

IV. PRIMATE Hardware Architecture

A. Overall PIM Architecture

We base PRIMATE hardware on HBM2E [14], as illustrated in Figure 4. We choose HBM as the basic architecture for two key reasons. First, HBM connects stacked DRAM dies using a large number of through-silicon-vias (TSVs), providing high internal/external bandwidth. Second, the stacked memory chips in HBM provide high area efficiency that supports the integration of in-memory logic to extend the functional flexibility of PIM [11]. We assume the HBM stack supports in-memory bit-serial processing while adopting near-bank units for reduction operations and inter-bank interconnect, as bit-serial design enables the most memory bandwidth among various PIM architectures [3]. To support dynamic token pruning, we propose a channel-level top-k mechanism in HBM. Specifically, it consists of multiple Top-k Engines (TE), each connecting to a channel IO link. The channel-level TE exploits the internal bandwidth and reserves memory space in each channel for parallel in-memory top-k selection. We also propose a new processing flow to fully utilize the top-k selection architecture for the PIM-accelerated Transformer.

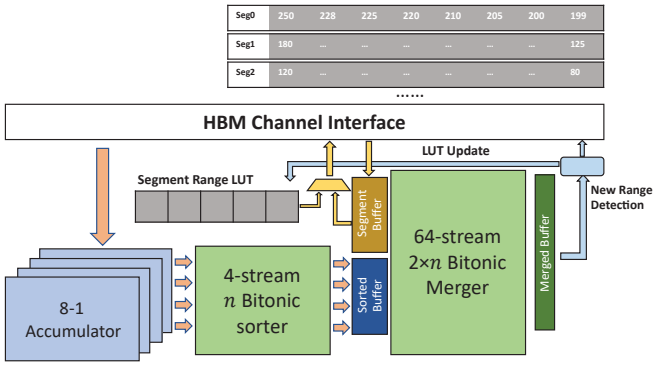

Fig. 5: Processing flow of in-memory top-k token selection.

B. Top-k Engine

We design our TE for minimal space overhead. The three key components: accumulators, segmented top-k buffer, and bitonic sorter and merger are illustrated in Figure 5. Each TE utilizes reserved memory (4 KB) in the corresponding channel as the Sorted Buffer for storing the top k values.

1) Accumulator: As introduced in Section III-A, each bank stores the attention scores (after multi-head softmax) for a subset of tokens. However, calculating the importance score of a token involves all tokens in this layer. In other words, each bank stores partial sums of attention importance for all tokens. With the current state of the art [27], we can use the bank-level reduction logic to reduce partial sums into one vector of size N, with N referring to the total token count in a layer. However, we still need to accumulate partial sums from different banks before sorting. Therefore, we add several accumulators in TE that fully utilize the channel-level bandwidth. Specifically, each accumulator receives 8 values from 8 banks and reduces to 1 value. For HBM2E, the channel-level interface sends 256b for each cycle. Therefore, we add 4 accumulators in each channel to support 8b partial sums to fulfill the bandwidth. Our investigation shows that 8b TE provides sufficient precision to retain the pruning pattern and the model is robust to small changes in token-dropping schemes.

2) Segmented Top-k Buffer: For each channel-level TE, PRIMATE reverses a memory space in the channel as the buffer for the sorted top-k values. Each buffer is divided into multiple *segments* in order to reduce the overhead of updating the buffer, as shown in Figure 5. This is because the limited channel bandwidth cannot support reading all values for top-k mechanisms in one cycle. Therefore, we need to sequentially read out a small set of values and progressively update the buffer. By using the buffer segmentation with a segment range look-up table, we can locate the segment for each insertion and only

update the buffer at the segment granularity, avoiding costly insertion on all values. The operations on the Sorted Buffer are normal memory operations that utilize data IO in HBM.

3) Bitonic Sorter and Merger: TE utilizes a hybrid strategy where it first generates a small sorted segment of size n using a parallel bitonic sorter, merges the segment with current top k tokens, and inserts it back into the buffer. The top k tokens are stored in equally sized segments where each segment has n tokens. Every insertion will update n tokens and automatically remove the last segment. Insertion by segment ensures regular data access pattern to the HBM channel. The 4-stream n bitonic sorter and 64-stream bitonic merger are generated using Spiral Project [29] and synthesized with Synopses Design Compiler.

C. Processing Flow of Top-k Token Selection

With the token-based dataflow, the self-attention layer generates attention scores between all tokens. For each token, the attention layer generates N attention scores where N denotes the total number of tokens. Therefore, each bank stores $t \times N$ attention scores, where t is the number of tokens allocated in the bank. These attention scores after the multi-head softmax operation are then accumulated to prepare for top-k selection.

HBM2E channels are 256-bit wide, and we use 8-bit importance scores as in TransPIM [27]. As shown in Figure 5, every cycle, 32 tokens are accessed and fed into 4 parallel 8-1 accumulators. Then, a 4-stream n bitonic sorter consumes 4 accumulated tokens every cycle. A sorted list with size n in Sorted Buffer is produced every $\frac{n}{4}$ cycle. Next, the first token in the Sorted Buffer is compared with the current ranges of each segment stored in a local Look Up Table (LUT). Thus, only segments containing tokens smaller than Sorted Buffer will be accessed sequentially and merged with the current Sorted Buffer, and the Sorted Buffer is automatically dropped if all segments have larger values than Sorted Buffer. A 64-stream $2 \times n$ bitonic merger is used to fully saturate HBM channel bandwidth for all accesses. Before writing back the newly merged segments to HBM, a range detection unit scans the addresses and updates the new range for each segment in the local LUT. If tokens for a layer are distributed across multiple channels, we use accumulators of multiple TEs to generate partial importance scores. Then, one of these TEs generates the final attention importance scores and merges the top-k values.

V. PRIMATE Pipeline Optimization

A. Pipeline Overview

To efficiently utilize available memory resources together with nonuniform layer sizes from token pruning, we propose a pipeline design to massively improve throughput by leveraging inter-layer parallelism. PRIMATE pipeline enables layers from multiple inferences on different inputs to run simultaneously on pre-allocated memory partitions with adequate sizes. The pipeline throughput is determined by the critical layer with the longest runtime. Specifically, for a particular time step τ in the pipeline, layer i is processing input sequence IN_τ. Upon finishing, it forwards the hidden outputs to layer $i+1$ to continue processing the input sequence IN_τ, while layer i receives its input for the next input sequence $IN_{\tau+1}$. After the cold start, the pipeline is capable of producing one inference every time step. By allocating partitions with appropriate sizes for different layers in the Transformer model, the pipelining approach evades the memory under-utilization problem caused by token pruning.

B. The Partition Problem

The configuration of the memory partitions dictates the performance of the pipeline. To start, we consider two naive approaches to the partition problem. First, we can equally distribute memory banks to all Transformer layers. Second, we consider a weighted partitioning based on the number of tokens (after pruning) in different layers. For simplicity, we denote them as NP1 and NP2 (Naive Pipeline). Through experiments, neither of the two methods comes close to optimal. The primary factor affecting the performance of a particular partition is a trade-off between the *token density*, i.e. the number of tokens per bank, and the *token quantity*, i.e. the total number of tokens the partition needs to process. Qualitatively speaking, high token density leads to dense memory mapping that reduces the data movement distances during computation, but it comes with higher compute latency as the same compute units are faced with increased load. This suggests a trade-off space between data movement cost and compute cost for a given workload and memory. However, considering a memory bank can hold tens of tokens, the design space is immense. To address the design space challenge, we propose the PRIMATE Optimization Framework (PrimateOpt) that can swiftly navigate to an efficient solution for given configurations of memory, workload, and pruning scheme.

C. PRIMATE Optimization Framework (PrimateOpt)

PrimateOpt consists of three optimizations: *Layer-wise Exploration*, *Global Adjustment*, and *Layer Merging*, denoted as PO1–3 (Primate Optimization). PrimateOpt is illustrated by Figure 6.

1) Layer-wise Exploration, PO1: The global optimization for pipeline partitioning introduces an immense design space that cannot be solved efficiently. In this stage, we perform design space exploration at the individual layer level: the framework analyzes the trade-off between token density and quantity, and subsequently determines the optimal tokens per bank and partition size (number of banks) for the layer to achieve optimal performance. Then, we can construct the partition scheme for the whole pipeline by sequentially allocating the explored partition sizes. We denote this scheme as PO1, which offers significantly better throughput than NP1 and NP2.

2) Global Adjustment, PO2: However, PO1 introduces a series of issues, the first of which is the cross-stack placements of some partitions, resulting in expensive data movements that need to cross HBM stacks. By concatenating optimal partition sizes from layer-wise exploration, certain partitions may be placed across two stacks when they can, in fact, fit within one. We mitigate this problem by performing stack alignment. To start, the framework identifies partitions that unnecessarily cross stack boundaries by comparing their partition size to the stack size. Then, PrimateOpt performs stack alignment by shifting the partition and the following ones forward (towards higher indexed banks) until it is aligned with the next stack. However, this shifting operation introduces unmapped memory between the aligned position of the partition and where it started from. Here, PrimateOpt searches for suitable one or more layers whose combined size best fills the gap, and then rearranges the found partition(s) to fill the gap to avoid the under-utilization side effect of stack alignments. Algorithmically, this is an iterative process that guarantees no unnecessary stack misalignment exists upon exiting.

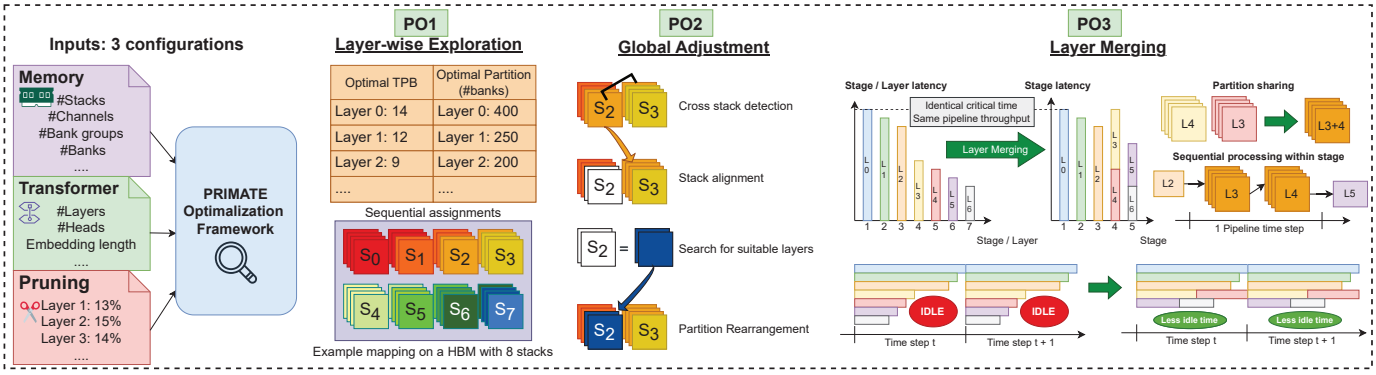

Fig. 6: Overview of the PRIMATE Optimization Framework

TABLE I: Latency and energy values of HBM2E.

t_{RC}	t_{RCD}	t_{RAS}	t_{FAW}	e_{ACT}	e_{LSA}	e_{GSA}	e_{IO}
45 ns	16 ns	29 ns	12 ns	909 pJ	1.51 pJ/b	1.17 pJ/b	0.80 pJ/b

Partition rearrangement results in no penalty when other partitions can fill the gaps exactly. We denote the resulting partitioning scheme as PO2.

3) Layer Merging, PO3: Although PO1 and PO2 provide solutions to spatial under-utilization through pipelining, the differences between layer-wise runtimes due to progressive token pruning cause temporal under-utilization of memory. As visualized in Figure 6, when a faster layer finishes, it needs to stay idle waiting for the critical, slower layer in the pipeline to finish. Only until then, layers can forward their hidden output to their respective next, and the pipeline progresses by a time step. To mitigate the induced temporal under-utilization, we propose Layer Merging that combines more than one layer into a single pipeline stage, denoted PO3. As illustrated in Figure 6, the combined layers are still processed sequentially in the same memory partition. We use the partition that originally belongs to the largest layer for the combined stage to ensure optimal layout for all. The only constraint for Layer Merging is that they are consecutive layers in the Transformer model and exhibit sequential dependency. We note that Layer Merging does not affect overall throughput if critical time does not change. Thus, Layer Merging improves the space efficiency, or throughput per GB, of the PRIMATE pipeline.

PrimateOpt is only run once before deployment for a given configuration. The related controls are translated into routines and are executed at runtime: between timesteps, memory instructions are issued to handle data forwarding and ensure the forward progress of the pipeline.

VI. Results

A. Experiment Setup

1) Architecture Configuration and Simulation: We use the HBM2E architecture with 10nm technology [14] as the hardware platform for PRIMATE. We assume a configuration of 8 stacks, each of which has 16GB of capacity with 16 physical channels. Each HBM2E channel supports 256b data links to read data out from 16 banks in the channel. Banks of a channel are grouped into 2 bank groups, each supporting 256b internal data link. Each bank reads data from subarrays to the bank sense amplifiers via 256b data links. We set the internal memory frequency at 500MHz and use the latency and energy values from previous work [15]. For simulation, we translate the Python implementation of Dynamic Token-pruning Transformer into memory commands, including normal memory read/write, in-memory computation, and near-memory processing. The commands are then passed into an in-house memory simulator that is similar to Ramulator [10]. The simulator integrates latency and energy values for different components that are either extracted from published work [15] or validated by our circuit simulation, as shown in Table I. We pass the configurations of HBM memory, Transformer model, and pruning scheme to PrimateOpt, which generates the memory mappings to configure our simulator using various methods discussed in this work, as summarized in Table III.

We design all extra in-memory logic in Verilog and synthesize them with Synopsys Design Compiler. The near-bank processing and the near-channel Top-k Engines are synthesized at 32nm. We then scale the power and area results to 10nm using the scaling method of previous work [20]. We also consider the overhead caused by the difference between ASIC and DRAM processes [13] in the reported values of this work. Our implementations are functionally verified using Xilinx Vivado.

Baseline Comparison: We identify TransPIM [27] as our baseline due to its recentness and superior performance than other accelerators including GPU, TPU, other PIM accelerators [5], [13], and ASIC Transformer accelerators [4], [22]. For a fair comparison, we model TransPIM [27] logic in our circuit design environment based on published specifications and integrate it with our simulation using HBM2E configuration.

Workloads: We evaluate PRIMATE using 4 Transformer workloads, denoted W1 to W4. Among them, W1 and W2 are two ViT models [2] targeting fine-grained image classification [7], [21]; W3 is a BERT model [1] targeting sentiment classification [18]; and W4 is a RoBERTa model [28] used for hyperpartisan news long document classification [8]. Table II presents the details about these workloads. These models are trained with a technique incorporating arbitrary token drops and an evolutionary search process to enhance the adaptability of the model to token drops and explore optimal accuracy-efficiency trade-offs. Pruning-wise, W1 incorporates non-uniform token dropping resulting in 39% tokens remaining (sum of all layers), while W2 – W4 prune 20% tokens per layer. Models used for W1 to W4 achieve accuracy of 91.1%, 89.6%, 92.7%, and 87.1%, respectively, which are comparable to their baseline performance without pruning. We quantize the models to 8-bit [26] before mapping them to memory.

TABLE II: Workloads used for PRIMATE evaluation

Workload	Model	Dataset	Layers	Sequence len.
W1	ViT	Stanford Dogs	12	786
W2	ViT	CUB-200-2011	12	3137
W3	BERT	SST-2	12	128
W4	RoBERTa	Hyperpartisan News	12	4096

TABLE III: Notations for memory mapping schemes

BS1	Current SOTA (TransPIM [27])
BS2	TransPIM [27] with pruning and without aggregation (Sec. III-A)
BS3	TransPIM [27] with pruning and aggregation support (Sec. III-A)
NP1	Naive Pipeline: Equal partitioning of entire memory (Sec. V-B)
NP2	Naive Pipeline: Weighted partitioning by input size (Sec. V-B)
PO1	PrimateOpt: Layer-wise Exploration results (Sec. V-C1)
PO2	PrimateOpt: Global Adjustments (Sec. V-C2)
PO3	PrimateOpt: Layer merging (Sec. V-C3)

B. Comparison with Existing Transformer Accelerators

We evaluate PRIMATE using throughput (inference per second), space efficiency (throughput per GB of memory used), and energy efficiency (throughput per Joule). We present normalized metrics with regard to the current SOTA TransPIM [27] in Figure 8 and 9. It is notable that the PrimateOpt progressively improves these metrics through Layer-wise Exploration, Global Adjustment, and Layer Merging. Compared to baseline [27], the PRIMATE architecture achieves up to 30.6×, average 21× better throughput; up to 29.5×, average 18.9× better space efficiency, and up to 4.3×, average 3.8× better energy efficiency on W1 to W4. By beating TransPIM [27] with significant margins, PRIMATE also offers superior performance to other ASIC-based Transformer accelerators including A^3 [4], SpAtten [22], and GPU-based solutions.

C. Effect of In-Memory Top-k Engines

We first compare the performance of PRIMATE with SOTA [27] without the support of in-memory top-k selection. Both architectures execute the Transformer in a pipelined manner optimized by the PrimateOpt. The baseline PIM architecture [27] sends all attention scores to the host CPU (AMD EPYC 7742 @ 2.25 GHz) for accumulation and top-k calculation using C++ `std::sort`. Our experiment shows the PRIMATE in-memory sorting design can reduce sorting overhead from 9.2%, 25.2%, 2.8%, 5.4% (W1 - W4) down to an average of 0.14%. PRIMATE in-memory top-k engine can offer up to 90× better standalone sorting cost. We also find CPU-based sorting incurs much more overheads on workloads with large token counts and smaller vector lengths, justifying our architectural advantage on large models like the ever-growing LLMs.

D. Effect of PrimateOpt

In this section, we refer to Figures 7 – 9 to show PRIMATE improvements. We evaluate layerwise runtime, throughput, space efficiency, and energy efficiency of different methods in Table III.

1) Layer-wise Exploration: Introduced in V-C1, token density and token quantity for a partition are both vital parameters to the latency of a layer, as they introduce a trade-off space between data movement cost and compute cost. In PRIMATE, we use Layer-wise Exploration to efficiently determine the optimal partition size for all layers and sequentially construct a pipeline. Figure 7 presents the per-layer latency of various partitioning schemes, whose details are present in Table III. From our experiments, compared to the naive equal partition and weighted partition schemes (NP1 and NP2), Layer-wise

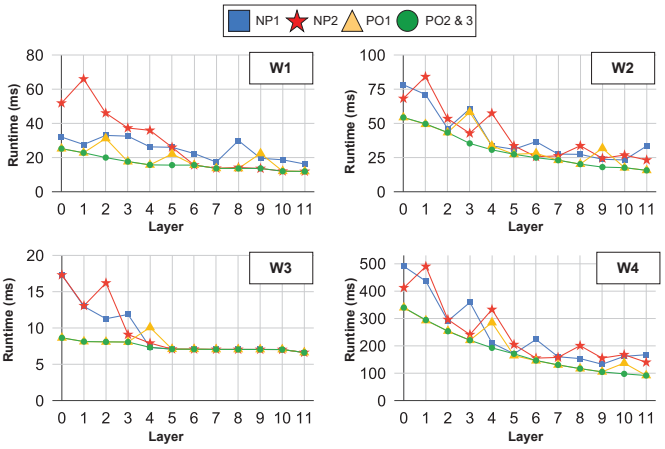

Fig. 7: Layerwise runtime comparison of different Partitioning Schemes. The overhead of configuring the layer-specific memory layout is included.

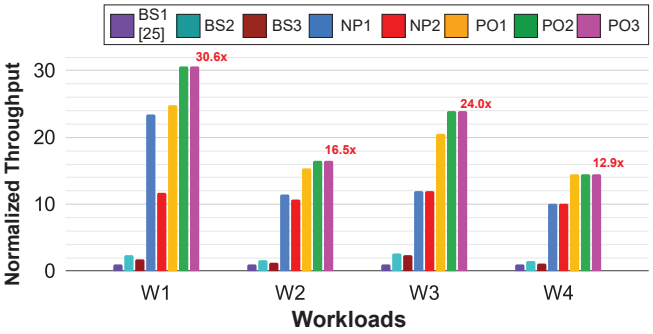

Fig. 8: Throughput of PRIMATE normalized to baseline. Value in red shows final improvement after PO3.

Exploration (PO1) offers an average 1.67× and up to 2.11× reduction in critical runtime in the pipeline for W1 to W4, respectively. Additionally, through exploration, PO1 offers better performance at lower memory usage, saving up to 46.7% of memory bank usage compared to the naive schemes. From progressive pruning, we expect a monotonically declining per-layer runtime throughout the model. We observe that PO1 is much closer to this goal than NP1 and NP2.

2) Global Adjustment: The performance of partitions derived by layer-wise exploration may not hold due to the placements of other partitions as we construct the pipeline sequentially. In Figure 7, the PO1 curve notably spikes on certain layers due to their cross-stack placements, which introduces costly cross-stack traffic. Global Adjustment performs stack alignment and partition rearrangement to eliminate cross-stack partitions while maintaining utilization. Figure 8 shows that compared to PO1, PO2 brings 23.7%, 7.0%, 16.7%, and 0% throughput improvement on W1 to W4, respectively. For the partition placed cross-stack, Global Adjustment can provide up to a 64.3% reduction in latency in tested workloads. Cost-wise, Global Adjustment does not incur any additional memory usage on W1 and W3, as other layers can exactly fill the gaps created by stack alignment. W2 and W4 receive small, 5.3% and 4.1% memory usage increase from stack alignment, respectively, for not finding exact matches due to other partition sizes. Note that for W4, its first layer happens to be the critical layer that cannot benefit from stack alignment. Yet, Global Adjustment is still an important

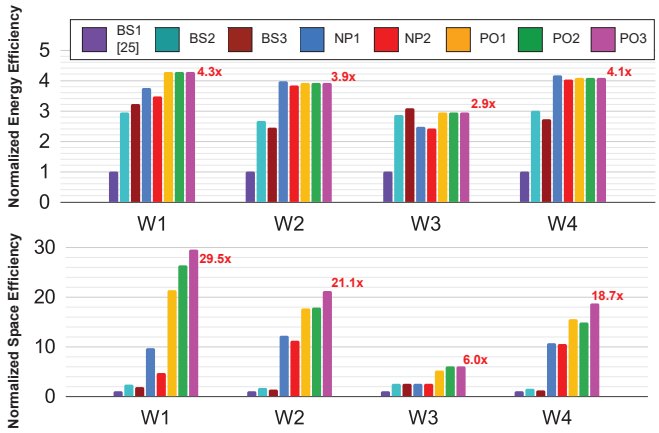

Fig. 9: Normalized space efficiency (inferences / second / GB) and normalized energy efficiency (inferences / second / J). Value in red shows final improvement after PO3.

prerequisite for subsequent optimization. After this stage, we observe the monotonically declining per-layer runtime throughout the model, a sign of efficient memory partitioning.

3) Layer Merging: Progressive pruning results in differences in layer runtimes that lead to stalls on the faster partitions, causing temporal under-utilization of memory as faster partitions await slower ones. We propose Layer Merging to allow one partition to sequentially process multiple faster layers, such that the combined runtime does not hurt critical time. Therefore, through Layer Merging, PRIMATE achieves the same throughput (same layerwise runtime in Figure 8) with less memory, thus a higher space efficiency. Our experiment in Figure 9 shows that Layer Merging can bring 11.7%, 18%, 0%, and 26.2% better throughput per GB compared to PO2. PrimateOpt algorithmically evaluates Layer Merging for given workloads and whether it should be applied: for example, in W3, merging any two layers results in a worse critical time that offsets the benefits. At this point, we achieve the best throughput and efficiency with PrimateOpt.

E. Overhead

The area for top-k engines (16 channel-level sorters) is 2.53 mm^2, 2.3% of one HBM2E stack area [14], and the power is 1.3W. We note that there is no impact on HBM capacity because the area overhead is significantly less than the 25% threshold evaluated in [5]. The optimizations proposed by PrimateOpt incur additional overhead in configuring the layer-specific memory layouts and data forwarding using memory instructions generated before runtime. At runtime, the memory routines are run between each pipeline time step. We carefully model these overheads and include them in our evaluations.

VII. Conclusion

We propose PRIMATE, a software-hardware co-design framework that synergizes token-pruning and PIM to accelerate Transformer models using HBM-based architecture. We evaluated PRIMATE using 4 representative Transformer workloads and observe up to 30.6× improvement in throughput, 29.5× improvement in space efficiency, and 4.3× better energy efficiency compared to the current state-of-the-art PIM accelerator for Transformers [27].

Acknowledgment

This work was funded by PRISM, one of seven centers in JUMP 2.0 (an SRC program sponsored by DARPA), SRC Global Research Collaboration (GRC) grant, and NSF grants #2112167, #2003279, #2100237, #2112665, #2008365, and #2052809.

References

[1] J. Devlin *et al.*, "BERT: pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019.
[2] A. Dosovitskiy *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," in *ICLR*, 2021.
[3] N. Hajinazar *et al.*, "Simdram: A framework for bit-serial simd processing using dram," in *ASPLOS*, 2021.
[4] T. J. Ham *et al.*, "A^3: Accelerating attention mechanisms in neural networks with approximation," in *HPCA*, 2020.
[5] M. He *et al.*, "Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning," in *MICRO*, 2020.
[6] M. Imani *et al.*, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *ISCA*, 2019.
[7] A. Khosla *et al.*, "Novel dataset for fine-grained image categorization," in *CVPR*, 2011.
[8] J. Kiesel *et al.*, "SemEval-2019 task 4: Hyperpartisan news detection," in *SemEval*, 2019.
[9] Y. Kim *et al.*, "A case for exploiting subarray-level parallelism (salp) in dram," in *ISCA*, 2012.
[10] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, 2015.
[11] Y.-C. Kwon *et al.*, "25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications," in *ISSCC*, 2021.
[12] C. Lee *et al.*, "Token adaptive vision transformer with efficient deployment for fine-grained image recognition," in *DATE*, 2023.
[13] S. Li *et al.*, "Drisa: A dram-based reconfigurable in-situ accelerator," in *MICRO*, 2017.
[14] C. Oh *et al.*, "22.1 a 1.1v 16gb 640gb/s hbm2e dram with a data-bus window-extension technique and a synergetic on-die ecc scheme," in *ISSCC*, 2020.
[15] M. O'Connor *et al.*, "Fine-grained dram: Energy-efficient dram for extreme bandwidth systems," in *MICRO*, 2017.
[16] Y. Rao *et al.*, "Dynamicvit: Efficient vision transformers with dynamic token sparsification," *NeurIPS*, 2021.
[17] V. Seshadri *et al.*, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *MICRO*, 2013.
[18] R. Socher *et al.*, "Recursive deep models for semantic compositionality over a sentiment treebank," in *ACL EMNLP*, 2013.
[19] S. Sridharan *et al.*, "X-former: In-memory acceleration of transformers," 2023. [Online]. Available: http://arxiv.org/abs/2303.07470
[20] A. Stillmaker *et al.*, "Toward more accurate scaling estimates of cmos circuits from 180 nm to 22 nm," *VLSI Computation Lab, ECE Dept., UC Davis, Tech. Rep. ECE-VCL-2011-4*, vol. 4, p. m8, 2011.
[21] C. Wah *et al.*, "The caltech-ucsd birds-200-2011 dataset," 2011.
[22] H. Wang *et al.*, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *HPCA*, 2021.
[23] J. Wang *et al.*, "A 28-nm compute sram with bit-serial logic/arithmetic operations for programmable in-memory vector computing," *JSSC*, 2019.
[24] A. Yazdanbakhsh *et al.*, "Sparse attention acceleration with synergistic in-memory pruning and on-chip recomputation," in *MICRO*, 2022.
[25] H. Yin *et al.*, "A-vit: Adaptive tokens for efficient vision transformer," in *CVPR*, 2022.
[26] A. H. Zadeh *et al.*, "Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference," in *MICRO*, 2020.
[27] M. Zhou *et al.*, "Transpim: A memory-based acceleration via software-hardware co-design for transformer," in *HPCA*, 2022.
[28] L. Zhuang *et al.*, "A robustly optimized BERT pre-training approach with post-training," in *Proceedings of the 20th Chinese National Conference on Computational Linguistics*, 2021.
[29] M. Zuluaga *et al.*, "Streaming sorting networks," *ACM TODAES*, 2016.