LATTIX: A Sparse Processing-In-Memory Accelerator for Speculative LLM Inference

Seungkwan Kang, Hyungseok Ko, Hyun Dong Lee, Myoungsoo Jung
Computer Architecture and Memory Systems Laboratory, KAIST

***Abstract*—In on-premise LLM inference, speculative decoding breaks the memory wall induced by the low batch size, by combining fast generation from a lightweight draft model with parallel verification by the target model. However, speculative decoding, by design, poses a dilemma in the draft model's size, where scaling up the draft model increases the per-token accept rate while simultaneously increasing draft latency, ultimately degrading end-to-end throughput. Processing-in-memory (PIM) architectures were designed to accelerate memory access through high internal bandwidth, yet their compute resources are severely limited by DRAM process area/power constraints. Pruning methods such as 2:4 semi-structured pruning alone cannot mitigate the compute bottleneck. Based on the observation that LLM activations are inherently near-sparse, we propose a *dual pruning scheme* that prunes both weights and activations in a 2:4 manner to significantly reduce the total required computation while preserving accuracy, along with a dedicated accelerator module, LATTIX, that efficiently executes this within PIM. To this end, we propose three runtime activation pruning methods suited to PIM's limited compute capability, a lightweight multiply-accumulate (MAC)-based compute engine consisting of a lookup-based selector logic, and pipelining and reordering techniques that maintain high utilization. Evaluated on two LLM families, LATTIX achieves 2.59× and 1.90× end-to-end speedup over dense and weight-only pruned draft models, respectively, without loss of accuracy.**

I. INTRODUCTION

As the service cost and data privacy concerns of large language models grow, *on-premise deployments*, where users deploy LLMs locally, are rapidly increasing in applications such as healthcare [1]–[3], finance [4]–[6], and personal assistants [7]–[9], eliminating external data transmission. Due to the extremely low batch sizes [10], [11], the compute time of on-premise deployments is dominated by general matrix-vector multiplication (GEMV), resulting in extremely low arithmetic intensity [12]–[14]. Consequently, the majority of the accelerator's compute units remain idle waiting for memory transfers, causing an order-of-magnitude gap between peak FLOPS and achievable FLOPS [10], [15], [16]. To address this inefficiency, *speculative decoding* [17], [18], in which a lightweight draft model rapidly generates multiple tokens in succession and an expensive target model verifies them in a single pass, has become the standard technique for local LLMs [19], [20]. This allows multiple tokens to be finalized per target invocation, amortizing the cost analogous to increasing the batch size. In speculative decoding, the *speculation length*, or the expected number of accepted tokens per target invocation, is a critical factor determining end-to-end throughput [17], [19].

Therefore, prior work has focused on accelerating the draft model's inference itself while preserving the speculation length, thereby achieving higher end-to-end throughput at the same target invocation frequency. Accordingly, prior approaches divide into two main directions. On the hardware side, *processing-in-memory (PIM)* accelerators place compute units within the DRAM banks to exploit high internal bandwidth, physically resolving the data movement bottleneck that dominates inference [21], [22], [22], [23]. On the software side, *model pruning* techniques retain only a subset of important weights, masking the rest and computing only on the retained elements, thereby reducing both memory and compute demands [24]–[26]. A representative example is 2:4 semi-structured pruning [27], [28], which removes the two smallest-magnitude elements from each group of four consecutive elements (hereafter *tile*) and stores and reads only the remaining two. However, both directions focus on memory bandwidth utilization and overlook the compute-side problem that arises in a PIM environment. PIM accelerates memory access through high internal bandwidth, yet its compute resources are severely limited by DRAM process area/power constraints [21], [29]. While pruning reduces both weight read volume and computation, the compute reduction from weight-only pruning alone remains insufficient to overcome PIM's severe compute scarcity. Ultimately, the draft model's inference becomes compute-bound, preventing PIM from utilizing its massive internal bandwidth.

Based on the observation that activations in LLM computation are inherently near-sparse, we propose *dual-side pruning*, where we apply 2:4 semi-structured pruning to both weights and activations. We then propose a dedicated accelerator module, LATTIX, that efficiently executes such a pruning scheme within PIM. Dual-side 2:4 pruning significantly reduces the actual required computation at the intersection of weight and activation tiles, providing additional compute reduction that weight-only pruning cannot achieve. This can therefore fundamentally resolve PIM's compute bottleneck and enable full utilization of the DRAM's internal bandwidth. LATTIX is designed as a lightweight compute engine to operate within the DRAM's area and power constraints, along with pipeline and tile reordering techniques that maintain high utilization despite the computation requirement varying over time. Evaluated on two real-world frontier model LLM families, LATTIX achieves an average of 2.59× and 1.90× end-to-end speedup over dense and weight-only pruned draft models, respectively.

The contributions of this work are as follows.

Dual-side 2:4 pruning. We analyze the compute-bound problem arising in speculative decoding on PIM-based architectures, and resolve it by applying 2:4 pruning to both weights and activations, which we refer to as dual-side 2:4 pruning, to fundamentally reduce computation. We propose 3 methods for pruning the activations at runtime via magnitude, while varying their hardware cost.

Lightweight accelerator. We first exploit the property that all types of intersection patterns require at most two multiply-accumulate (MAC) operations per tile, to design a compute engine consisting of only two MAC units that can consume a single whole tile per cycle. However, the two MAC units require routing logic that can direct the corresponding non-zero data to the compute engine. We exploit the property that the number of possible pattern types is highly limited, adopting a memoization-based logic to minimize resource usage.

Tile pipelining. In dual-side sparsity, each tile pair (weight × activation) requires a variable number of MAC operations from zero to two. We propose two techniques to maximize the hardware utilization. First, we skip tile pairs that require no computation at all. Second, we propose a pipeline structure that issues two tile pairs simultaneously at every cycle, processing both tiles in a single cycle

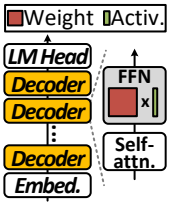


Fig. 1: Transformer.

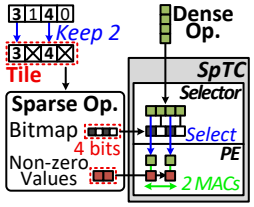


Fig. 2: Semi-structured pruning.

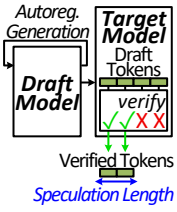
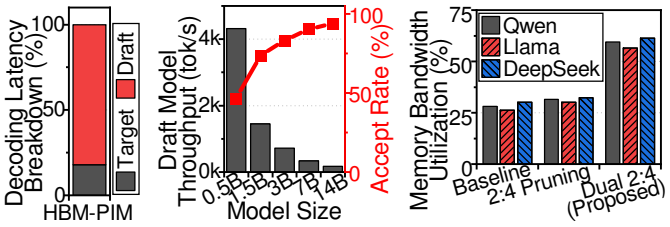


Fig. 3: Speculative decoding.



(a) E2E breakdown. (b) Draft model dilemma. (c) Compute wall.

Fig. 4: Speculative decoding.

when their combined workload fits within the 2-MAC capacity.

Runtime tile reordering. Since the computation order among tiles in an inner product does not affect the result due to commutativity, each PIM module computes the tile pair's overlap count at runtime and rearranges the order so that tile pairs with low and high computation demands are placed adjacent to one another.

II. Background

Transformer architecture. Figure 1 illustrates the structure of a transformer decoder. A transformer consists of an embedding layer, multiple decoder layers, and an LM head [30], where each decoder layer comprises a self-attention sub-layer and a feed-forward network (FFN) sub-layer. The key factor determining the computational characteristics of LLM inference is the *batch size*, which is the number of requests processed simultaneously. In low-batch scenarios, the FFN sub-layer dominates the computation time [10], [13], and its core operation is the multiplication of a weight matrix and an activation.

2:4 semi-structured pruning. 2:4 semi-structured sparsity groups every four consecutive elements into a *tile* and retains exactly two non-zero elements per tile (Figure 2). Existing 2:4 pruning techniques [24], [25], [27] inject this pattern into model weights, improving inference performance while maintaining high accuracy. Each pruned tensor is stored as a value tensor containing only the non-zero values and a bitmap recording the non-zero positions within each tile (4 bits per tile). The compressed format reduces both memory traffic and storage by nearly half. The Sparse Tensor Core (SpTC), introduced in the NVIDIA Ampere [31] architecture, leverages this compressed format to accelerate matrix multiplication between a (2:4) sparse operand and a dense operand at the hardware level. The internal structure of SpTC augments a conventional MAC-based processing element (PE) with a selector unit at the front end (Figure 2). The non-zero values from the sparse operand are fed to the PE, together with the bitmap. The selector then consults the bitmap to read the corresponding element from the dense operand and forwards it to the MAC unit. As a result, only two MACs are performed per tile instead of four, enabling SpTC to process twice as many tiles in the same number of cycles, achieving a theoretical 2× throughput improvement.

Speculative decoding. Speculative decoding is a technique that leverages a lightweight *draft model* to increase the token throughput of LLM inference [17], [18]. In speculative decoding, the draft model autoregressively generates k tokens, after which the target model verifies them in a single forward pass (Figure 3). Specifically, by appending the k tokens pre-generated by the draft model to the prompt and feeding them to the target model, the target model can simultaneously generate outputs for all k positions. The target model's outputs are sequentially compared with the draft token at each position and only correct drafts are considered verified and accepted. Note that, due to the autoregressive constraint, all draft tokens after the first mismatch are rejected.

The average number of tokens accepted per target model invocation is called the *speculation length*, which represents the effective inference bandwidth of the target model. Prior work has proposed tree-based [32], [33] or graph-based [34], [35] speculation structures. In such structures, the draft model generates multiple token candidates at each position rather than a single token. This can increase the speculation length by widening the verification scope, but at the cost of increased draft model computation.

III. Motivation

End-to-end breakdown. Throughout this section, we analyze speculative decoding on a system where the draft model runs on an HBM-PIM device and the target model on two A100 GPUs; the detailed hardware and software configurations are summarized in Table I. Figure 4a shows the fraction of the total decoding latency spent on the target and draft model computations. With a large speculation tree ([4,2,2,1,1]), the draft model computation consumes 82.1% of the total decoding latency.

Draft model dilemma. The end-to-end throughput of speculative decoding is governed by two conflicting factors that both depend on the draft model size: a larger draft model approximates the target model's output distribution more closely, raising the per-token accept rate and thus the expected speculation length, but its slower inference lengthens every speculation iteration. Figure 4b quantifies this trade-off. We performed tree-based speculative decoding while sweeping the draft model size of the Qwen2.5 family [36] from 0.5B to 14B against a 72B target. As the draft model size grows, the accept rate rises monotonically from 47% to 94%, while the draft throughput collapses from 4.3k to 0.2k tokens/s. A small draft (0.5B) offers high throughput, but its low accept rate fails to amortize the cost of target verification. On the other hand, a large draft (14B) achieves a high accept rate, but the draft itself becomes the bottleneck and the end-to-end throughput degrades. No point on the model-size axis therefore satisfies both the accept rate and the latency at once.

Compute wall in PIM. PIM architectures [21]–[23] have been proposed to accelerate the memory-intensive autoregressive inference of the large language model by accessing memory at the massive internal bandwidth of the DRAM banks. However, due to the area and power constraints of the DRAM process, their compute resources are extremely scarce, leaving the internal bandwidth largely underutilized. Figure 4c reports the memory bandwidth utilization of an HBM-PIM device under different pruning schemes. Without pruning, the dense baseline utilizes only 27–31% of the internal bandwidth across all of Qwen, Llama, and DeepSeek, meaning that the bottleneck is compute rather than memory bandwidth. Conventional weight-only 2:4 pruning cannot break this compute wall, as it reduces memory traffic and computation at a similar rate; the utilization stays nearly unchanged at 30–33%.

Dual-side 2:4 pruning. The preceding analysis shows that resolving PIM's compute bottleneck requires an additional compute reduction beyond weight pruning. We enable this with a dual-side sparse

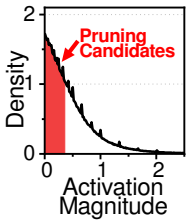
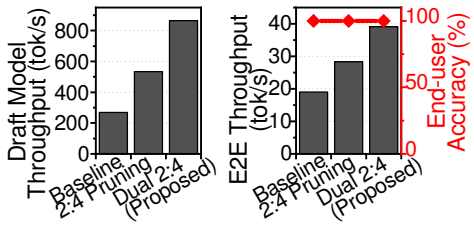
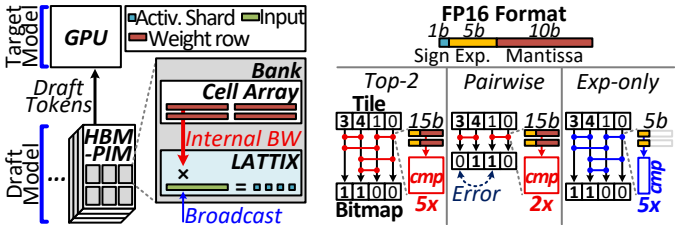


Fig. 5: Activation sparsity.



(a) Draft throughput. (b) E2E throughput.

Fig. 6: Dual 2:4 Pruning.



(a) System architecture. (b) Activation pruning.

Fig. 7: Overview of proposed system.

scheme, based on the observation that LLM activations are inherently near-sparse. Figure 5 shows the probability density of the activation magnitudes in the FFN layers. The elements are heavily concentrated in the near-zero region (shaded), where they contribute only marginally to the final output. This distribution arises because the gated activation functions (SwiGLU, GeGLU) standard in modern LLMs suppress all but a few channels. Consequently, 2:4 pruning that removes the two smallest-magnitude elements within each activation tile can greatly reduce the required computation while barely compromising inference accuracy. Exploiting this near-sparsity, the proposed dual 2:4 pruning applies 2:4 pruning to both the weights and the activations, so that only the elements surviving on both sides of each tile-pair inner product need to be computed. Even with the compute capability of the baseline HBM-PIM left unchanged, an idealized dual 2:4 execution that eliminates all the skipped MACs doubles the bandwidth utilization to 57–62% (Figure 4c), substantially relieving the compute wall. Figure 6a shows the theoretically achievable draft throughput under each pruning scheme. Conventional weight-only 2:4 pruning halves both the computation and the memory traffic, improving the draft throughput by 2.0×; additionally pruning the activations delivers a further 1.62×. Figure 6b translates these draft-side gains into the end-to-end throughput perceived by the user: weight-only 2:4 achieves a 1.49× gain over the dense baseline, and ideally, pruning the activations as well provides an additional 1.39×. Moreover, since target verification filters the draft's output and only the tokens the target model agrees with ever reach the user, the end-user accuracy remains identical regardless of the pruning scheme, as shown in Figure 6b.

IV. Design

System architecture. Figure 7a shows the overall system architecture. The system consists of a host processor (e.g., GPU) connected to a memory stack via an HBM interface. Each memory stack is a PIM device, with a lightweight compute logic in each bank, inspired by the Samsung HBM-PIM [21] architecture. Consequently, the overall system executes a heterogeneous dataflow in which the target model runs on the host processor and the draft model runs on the HBM-PIM. The draft model's weights are distributed across the banks in a row-wise fashion to exploit the high aggregated internal bandwidth. Each shard of an activation matrix is generated within the bank, as a result of a computation between the distributed weight matrix and input. The draft model's output tokens generated through repeated iterations are forwarded to the host processor for use in target model computation.

Activation pruning. Weight pruning can determine a fixed 2:4 mask in advance through offline calibration. However, activations are runtime tensors newly generated as the output of the previous layer at every token, making it impossible to determine the mask in advance. PIM's compute logic supports only simple comparisons and integer-level operations due to the area/power constraints of the DRAM process. Complex operations such as exponentials and division cannot be performed, so pruning methods must be designed within these constraints. We designed three methods that differ in the precision of comparison operations. Figure 7b illustrates each method. The first method is the most naive approach, selecting the actual magnitude-based top-2 elements. Although this can compute the exact largest values, it requires a selection network of up to 5 comparisons, resulting in a large comparator area.

We therefore designed two additional approximate methods that reduce the comparator cost in two distinct ways. The first method is *pairwise tournament*, which partitions the four elements within a tile into two fixed pairs (pos0 and pos1, pos2 and pos3) and selects the element with the larger magnitude from each pair. Unlike the previous method, this determines the 2:4 mask with only two comparisons, minimizing HW cost. However, if the two largest elements fall in the same pair, the second-largest element is pruned and a lower-ranked element is selected instead. The final method is *exponent-only comparison*. Like the top-2 method, it selects the top-2 from all four values, requiring 5 comparators. However, it minimizes the area and latency overhead of each comparator by extracting only the 5-bit exponent field and comparing them as unsigned integers. A comparative evaluation is presented in Section V.

Activation pruning can only be performed at runtime after computation completes. When the previous GEMV operation finishes, the activation resides in a distributed state across the local memory of each bank. We route this activation back into the compute logic, and perform tile-level pruning according to the methods described above. This epilog pruning can be performed in-place without a separate memory read, since the activation values already reside within the bank. Upon completion of pruning, a 4-bit bitmap encoding the non-zero positions and two non-zero values are generated in compressed format within each bank. This pruned activation is broadcast to all banks in tile units, serving as the input to the LATTIX module together with the weights for the next layer's GEMV operation.

LATTIX module. Figure 8 shows the detailed structure of each LATTIX compute module. Each compute module takes a weight tile and an activation tile as input, computes their products, and accumulates the results. Since both the weight tile and the activation tile are pruned in a 2:4 structure, performing a bitwise AND of the non-zero positions of the two tiles yields 0 to 2 overlapping positions. We term these overlapping positions the *intersection pattern*; multiplication of actual values is needed only at these positions, since at the remaining positions at least one operand is necessarily zero and the product is therefore zero. Once the intersection is determined, the corresponding values must be located in the compressed vectors of the weight and activation and routed to the MAC unit as inputs.

To determine which data to read and how to compute based on the shape of the two bitmaps, each compute module's selector takes the two bitmaps as input and determines three control values. First, the operand selector is a multiplexer (MUX)-based control signal that determines which of the two non-zero values from the activation and

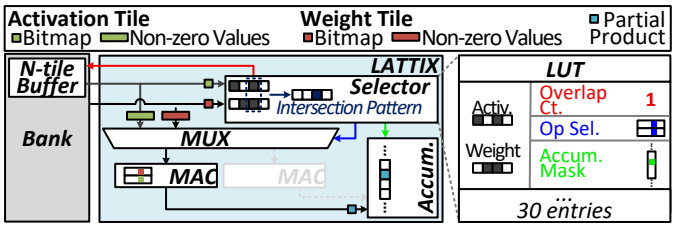
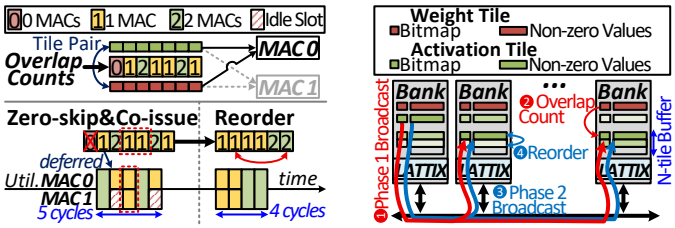


Fig. 8: LATTIX compute module.



(a) Execution pipeline. (b) Two-phase broadcasting.

Fig. 9: Execution pipeline and two-phase broadcasting.

weight, respectively, to route to which MAC. Second, the overlap count is used for pipeline scheduling, as described shortly, serving as the basis for tile co-issue and zero-skip decisions. Third, the accumulation mask is a control signal that selects the correct output position to which partial products should be accumulated, taking scheduling into account.

A naive selector logic would require complex computation between the two bitmaps. However, since the 2:4 patterns of the weight and activation each have only six possibilities (C(4,2) = 6), the total number of possible pair types is limited to thirty-six. Furthermore, six of these cases require no computation, so only thirty patterns are effectively valid. Moreover, because each pattern deterministically determines the operand routing, overlap count, and accumulation mask, there is no need for any data-dependent decisions at runtime. We therefore adopt a fixed lookup-table-based approach for the selector. The area of the resulting selector, a 30-entry combinational logic, requires fewer resources than that of a single FP16 multiplier.

Execution pipeline. As described above, each intersection pattern has at most two overlapping positions, and thus the LATTIX compute module consists of only two MAC units. Consequently, each compute module performs 0, 1, or 2 MAC operations per tile pair. However, since most (more than 80% on average) of the tile pairs require only 0 or 1 MACs, idle slots frequently occur relative to the 2-MAC capacity when processing a single tile pair at a time. To address this underutilization, we introduce three ideas (Figure 9a). The first is zero-skipping: tile pairs with an overlap count of 0 are immediately skipped through metadata inspection alone, without occupying a MAC unit. The second is tile co-issuing: tile pairs are read two at a time, and if the sum of their required MACs is two, both complete in a single cycle; if the sum exceeds two, the tile pair with the smaller MAC count is deferred to the next cycle. Such overflow-induced stalls depend on the MAC count combination of adjacent tile pairs; in particular, when 1-MAC and 2-MAC tile pairs are consecutive, stalls become frequent, worsening underutilization. The third is therefore tile reordering, which places 1-MAC tile pairs and 2-MAC tile pairs adjacent to each other rather than leaving them interleaved.

To realize these ideas, we adopted two-phase broadcasting and N-tile buffering in LATTIX (Figure 9b). Two-phase broadcasting splits the process of broadcasting the activation results into a bitmap phase (Phase 1) and the non-zero value phase (Phase 2). In Phase 1, all bitmaps are broadcast to all the banks, and each bank intersects the received bitmaps with its own weight tile bitmaps to pre-compute the overlap count for every tile pair. Since each tile pair's overlap count was pre-computed in Phase 1, each bank can immediately skip tile pairs with an overlap count of 0 without using MAC units. In Phase 2, the activation's non-zero values are broadcast, starting from bank 0, in units of N tiles, and each bank buffers the received tiles in its private, dedicated tile buffer. This buffering allows each bank to perform random access within the N tiles, enabling tile-level reordering. Once processing of the N tiles is complete, a reduce operation confirms that all banks have finished, and the broadcast proceeds with the next N tiles.

Tile reordering exploits the fact that in the inner product, the order of tile pair computations does not affect the final sum, since the results are accumulated. The challenge is that activation tiles are broadcast sequentially from a central source so all banks receive them in the same order, yet each bank stores a different weight row, so the overlap count per tile pair differs across banks. Therefore, by leveraging the N-tile buffering described above, we enable each bank to locally and greedily rearrange the processing order of tile pairs within its buffer (Figure 9b), with inter-bank coordination occurring only at the granularity of N tiles. Note that the buffer size, N, is a design parameter that determines the trade-off between reordering quality and buffer die area.

V. EVALUATION

A. Evaluation Setup

Methodology. We evaluated LATTIX on a real system equipped with two NVIDIA A100 GPUs; the detailed hardware and software configurations are summarized in Table I. The target model is sharded across the two GPUs, while the draft model runs on a PIM device of three HBM stacks; all pruned configurations use the same device. We used two real-world model pairs, Qwen2.5-72B with a Qwen2.5-7B draft [36] and DeepSeek-67B with a DeepSeek-7B draft [44], on three datasets: GSM8K [45], BBH [46], and HotpotQA [47]. All draft models were fine-tuned after pruning: weights are pruned one-shot with Wanda [25], and the dual-side drafts are further fine-tuned via knowledge distillation to recover the token accept rate. Every configuration runs the same tree-based speculation protocol [32], a token tree with per-depth branching factors [4,2,2,1,1] (60 candidate tokens per iteration). We measured the target-side verification latency and the token accept rate (and the resulting speculation length) of every draft variant directly on this system. Since bank-level PIM silicon is not commercially available, we supplemented these measurements with an in-house cycle-level simulator that models the draft-side PIM device, following the published Aquabolt-XL specification [21]; the measured target-side latencies and accept rates are combined with the simulated draft-side latencies to obtain end-to-end throughput. To estimate the hardware cost of LATTIX, we implemented all of its components in Verilog RTL and synthesized them under the setup summarized in Table I, obtaining the area of each module. As PIM logic is fabricated in a DRAM process whose

TABLE I: Evaluated system.

HW Configurations		SW Configurations			
GPU	2× A100 80 GB [31]	**Ubuntu**	v22.04	**CUDA** [37]	v12.4
CPU	AMD EPYC 7763	**PyTorch** [38]	v2.9.1	**Transformers** [39]	v5.9.0
Mem.	DDR4 1 TB	**vLLM** [40]	v0.6.3	**Accelerate** [41]	v1.14.9

Area Estimation			
Synthesis	Yosys [42] v0.68	**Cell library**	NanGate FreePDK45 [43]
PIM clock	300 MHz	**Area budget**	iso-area w/ PCU [21]

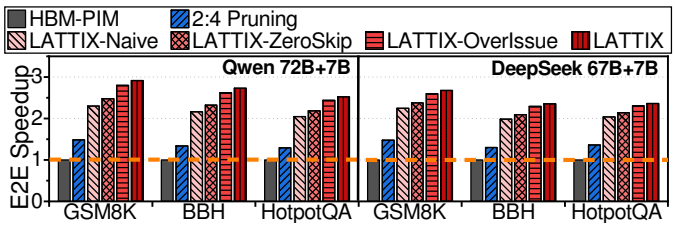
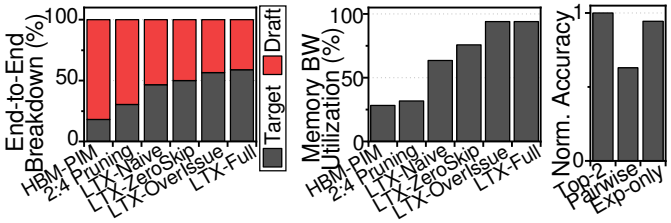


Fig. 10: Overall end-to-end speedup.



(a) E2E breakdown. (b) Memory utilization. (c) Pruning.

Fig. 11: Detailed evaluation.

cells are larger than those of a logic process, we use the synthesized areas as relative costs between modules; based on this accounting, all compared configurations are provisioned at equal area, with the LATTIX modules of each bank pair fitting within the footprint of the baseline's compute unit.

Compared configurations. We compared six configurations, all running the same draft models and speculation protocol on an equal-area PIM device. HBM-PIM is the baseline PIM modeled after Samsung Aquabolt-XL [21], which executes the dense draft on its per-bank-pair SIMD compute units. 2:4 Pruning augments each compute unit of Aquabolt-XL with an SpTC-style selector unit [27], [31], executing the weight-only 2:4-pruned draft; the pruned weights are skipped, but the activations remain dense. The remaining four configurations credit each LATTIX component separately: LATTIX-Naive replaces the compute units with the per-bank LATTIX modules (Section IV) and runs the dual-side 2:4 draft, but issues a single tile pair per cycle; LATTIX-ZeroSkip adds zero-skip; LATTIX-OverIssue adds tile co-issue; and LATTIX further adds runtime tile reordering, completing the full design. All LATTIX configurations generate the activation masks with the Exp-only pruning logic.

B. Evaluation Result

Overall Performance. Figure 10 reports the end-to-end speedup of the six configurations over the HBM-PIM baseline, across the two model pairs and three datasets. 2:4 Pruning doubles the drafting throughput by halving both the weight traffic and the MAC operations, yet delivers only 1.36× end-to-end, on average; since the compute reduction merely tracks the memory reduction, the draft remains compute-bound on PIM, and the accept-rate loss of weight pruning shortens the speculation length. LATTIX-Naive achieves 2.13× over the dense baseline and 1.57× over 2:4 Pruning, on average, by executing the dual-side 2:4 draft on the per-bank LATTIX modules: dual-side pruning halves the required MACs once more, while the bank-level placement provides twice the multipliers within the same area budget and exposes the full internal bandwidth of every bank. LATTIX-ZeroSkip discards the tile pairs whose bitmaps share no overlapping position without occupying MAC slots, adding 6% on average. LATTIX-OverIssue issues two tile pairs per cycle so that low-demand tile pairs no longer leave a MAC slot idle, adding another 11% on average. LATTIX further reorders the tile pairs within each bank's buffer, placing low- and high-demand tiles adjacent to each other to minimize co-issue stalls, and contributes a further 4% on average. As a result, LATTIX achieves an average end-to-end speedup of 2.59× and up to 2.91× (Qwen, GSM8K) over the dense baseline, consistently exceeding 2.3× across every model pair and dataset.

End-to-End Breakdown. Figure 11a decomposes the end-to-end inference time of the six configurations into target verification and draft computation. Under HBM-PIM, drafting consumes 82.1% of the total time, confirming that the draft model is the dominant bottleneck of tree-based speculation on PIM (Section III); 2:4 Pruning still spends 69.7% of its time on drafting. The LATTIX variants progressively shrink the draft share to 53.5%, 50.0%, 43.5%, and finally 41.1%, at which point drafting and verification are nearly balanced.

Memory utilization. Figure 11b shows the internal memory bandwidth utilization of the draft device for the six configurations. HBM-PIM sustains only 28.2% of the internal bandwidth, and 2:4 Pruning barely improves it to 31.8%, as both are throttled by the compute wall analyzed in Section III. LATTIX-Naive raises the utilization to 63.5% with its doubled multipliers and halved per-tile computation, and LATTIX-ZeroSkip pushes it to 75.9% by eliminating wasted cycles on zero-overlap tiles. LATTIX-OverIssue and the full LATTIX saturate the internal bandwidth at 94.1%, converting the draft execution from compute-bound to memory-bound. Note that tile reordering does not further increase the bandwidth utilization, which is already saturated; its benefit instead comes from higher MAC utilization at the larger batch sizes of the deeper tree levels, as reflected in the 4% end-to-end gain reported above.

Activation Pruning. Figure 11c compares the three activation-pruning logic designs (Section IV) that generate the 2:4 masks of runtime activations in LATTIX. We report the final token acceptance rate of the resulting draft model, normalized to that of the exact Top-2 selection. Pairwise Tournament is the cheapest yet retains only 62.9% of the acceptance rate. In contrast, the proposed Exp-only comparison achieves 94.5% of the Top-2 acceptance rate at less than half of its area, showing that exponent-level precision is sufficient to identify the dominant elements of gated activations.

VI. CONCLUSION

In this work, we resolve the compute-bound inference wall of on-premise speculative decoding in processing-in-memory (PIM) architectures. We propose dual-side 2:4 pruning, which applies 2:4 semi-structured pruning to both weights and activations of the draft model, and LATTIX, a dedicated PIM accelerator module that efficiently executes this pruning scheme. We propose a 2-MAC compute engine, utilizing the fact that only at most two MAC operations are needed per tile pair, together with a lightweight, lookup-based selector logic exploiting the limited number of valid intersection patterns. To maintain high compute utilization, we introduce a two-phase broadcasting scheme with zero-skipping, tile co-issuing, and local reordering techniques. Evaluated on two real-world LLM families, LATTIX achieves an average of 2.59× and 1.90× end-to-end speedup over dense and weight-only pruned draft models, respectively.

ACKNOWLEDGMENT

The authors thank anonymous reviewers for their constructive feedback. This work was supported by Samsung Research Funding & Incubation Center for Future Technology under Project Number SRFC-IT2302-05. This work is protected by one or more patents. Myoungsoo Jung is the corresponding author (mj@camelab.org).

References

[1] A. J. Thirunavukarasu, D. S. J. Ting, K. Elangovan, L. Gutierrez, T. F. Tan, and D. S. W. Ting, "Large language models in medicine," *Nature medicine*, vol. 29, no. 8, pp. 1930–1940, 2023.

[2] K. Singhal, S. Azizi, T. Tu, S. S. Mahdavi, J. Wei, H. W. Chung, N. Scales, A. Tanwani, H. Cole-Lewis, S. Pfohl *et al.*, "Large language models encode clinical knowledge," *Nature*, vol. 620, no. 7972, pp. 172–180, 2023.

[3] X. Yang, A. Chen, N. PourNejatian, H. C. Shin, K. E. Smith, C. Parisien, C. Compas, C. Martin, A. B. Costa, M. G. Flores *et al.*, "A large language model for electronic health records," *NPJ digital medicine*, vol. 5, no. 1, p. 194, 2022.

[4] S. Wu, O. Irsoy, S. Lu, V. Dabravolski, M. Dredze, S. Gehrmann, P. Kambadur, D. Rosenberg, and G. Mann, "Bloomberggpt: A large language model for finance," *arXiv preprint arXiv:2303.17564*, 2023.

[5] H. Yang, X.-Y. Liu, and C. D. Wang, "Fingpt: Open-source financial large language models," *arXiv preprint arXiv:2306.06031*, 2023.

[6] Y. Li, S. Wang, H. Ding, and H. Chen, "Large language models in finance: A survey," in *Proceedings of the fourth ACM international conference on AI in finance*, 2023, pp. 374–382.

[7] T. Gunter, Z. Wang, C. Wang, R. Pang, A. Narayanan, A. Zhang, B. Zhang, C. Chen, C.-C. Chiu, D. Qiu *et al.*, "Apple intelligence foundation language models," *arXiv preprint arXiv:2407.21075*, 2024.

[8] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love *et al.*, "Gemma: Open models based on gemini research and technology," *arXiv preprint arXiv:2403.08295*, 2024.

[9] Y. Song, Z. Mi, H. Xie, and H. Chen, "Powerinfer: Fast large language model serving with a consumer-grade gpu," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024, pp. 590–606.

[10] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Proceedings of machine learning and systems*, vol. 5, pp. 606–624, 2023.

[11] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley *et al.*, "Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–15.

[12] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[13] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 711–732, 2021.

[14] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "Flexgen: High-throughput generative inference of large language models with a single gpu," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 094–31 116.

[15] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 118–132.

[16] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve," in *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, 2024, pp. 117–134.

[17] Y. Leviathan, M. Kalman, and Y. Matias, "Fast inference from transformers via speculative decoding," in *International conference on machine learning*. PMLR, 2023, pp. 19 274–19 286.

[18] C. Chen, S. Borgeaud, G. Irving, J.-B. Lespiau, L. Sifre, and J. Jumper, "Accelerating large language model decoding with speculative sampling," *arXiv preprint arXiv:2302.01318*, 2023.

[19] H. Xia, Z. Yang, Q. Dong, P. Wang, Y. Li, T. Ge, T. Liu, W. Li, and Z. Sui, "Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding," *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 7655–7671, 2024.

[20] Z. Zhou, X. Ning, K. Hong, T. Fu, J. Xu, S. Li, Y. Lou, L. Wang, Z. Yuan, X. Li *et al.*, "A survey on efficient inference for large language models," *arXiv preprint arXiv:2404.14294*, 2024.

[21] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin *et al.*, "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)*. IEEE, 2021, pp. 43–56.

[22] W. Kim, Y. Lee, Y. Kim, J. Hwang, S. Oh, J. Jung, A. Huseynov, W. G. Park, C. H. Park, D. Mahajan *et al.*, "Pimba: A processing-in-memory acceleration for post-transformer large language model serving," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, 2025, pp. 292–307.

[23] Y. He, H. Mao, C. Giannoula, M. Sadrosadati, J. Gómez-Luna, H. Li, X. Li, Y. Wang, and O. Mutlu, "Papi: Exploiting dynamic parallelism in large language model decoding with a processing-in-memory-enabled computing system," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 766–782.

[24] E. Frantar and D. Alistarh, "Sparsegpt: Massive language models can be accurately pruned in one-shot," in *International conference on machine learning*. PMLR, 2023, pp. 10 323–10 337.

[25] M. Sun, Z. Liu, A. Bair, and Z. Kolter, "A simple and effective pruning approach for large language models," in *International Conference on Learning Representations*, vol. 2024, 2024, pp. 4942–4964.

[26] X. Ma, G. Fang, and X. Wang, "Llm-pruner: On the structural pruning of large language models," *Advances in neural information processing systems*, vol. 36, pp. 21 702–21 720, 2023.

[27] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius, "Accelerating sparse deep neural networks," *arXiv preprint arXiv:2104.08378*, 2021.

[28] A. Zhou, Y. Ma, J. Zhu, J. Liu, Z. Zhang, K. Yuan, W. Sun, and H. Li, "Learning n:m fine-grained structured sparse neural networks from scratch," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=K9bw7vqp_s

[29] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim *et al.*, "A 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2 tflops programmable computing unit using bank-level parallelism, for machine learning applications," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64. IEEE, 2021, pp. 350–352.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[31] NVIDIA, "NVIDIA A100 tensor core GPU architecture: Unprecedented acceleration at every scale," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, 2020, accessed: 2026-08-01.

[32] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, Z. Wang, Z. Zhang, R. Y. Y. Wong, A. Zhu, L. Yang, X. Shi *et al.*, "Specinfer: Accelerating large language model serving with tree-based speculative inference and verification," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 932–949.

[33] T. Cai, Y. Li, Z. Geng, H. Peng, J. D. Lee, D. Chen, and T. Dao, "Medusa: Simple llm inference acceleration framework with multiple decoding heads," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.

[34] Z. Gong, J. Liu, Z. Wang, P. Wu, J. Wang, X. Cai, D. Zhao, and R. Yan, "Graph-structured speculative decoding," in *Findings of the Association for Computational Linguistics: ACL 2024*, 2024, pp. 11 404–11 415.

[35] Z. Chen, A. May, R. Svirschevski, Y. Huang, M. Ryabinin, Z. Jia, and B. Chen, "Sequoia: Scalable and robust speculative decoding," in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 129 531–129 563. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2024/file/ea1f5f0878d43ff4fb8bf64ef4a2326c-Paper-Conference.pdf

[36] Qwen, :, A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, H. Lin, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Lin, K. Dang, K. Lu, K. Bao, K. Yang, L. Yu, M. Li, M. Xue, P. Zhang, Q. Zhu, R. Men, R. Lin, T. Li, T. Tang, T. Xia, X. Ren, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Wan, Y. Liu,

Z. Cui, Z. Zhang, and Z. Qiu, "Qwen2.5 technical report," 2025. [Online]. Available: https://arxiv.org/abs/2412.15115

[37] NVIDIA, "CUDA C++ programming guide," https://docs.nvidia.com/cuda/cuda-programming-guide/index.html, 2025, accessed: 2026-08-01.

[38] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[39] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, 2020, pp. 38–45.

[40] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th symposium on operating systems principles*, 2023, pp. 611–626.

[41] S. Gugger, L. Debut, T. Wolf, P. Schmid, Z. Mueller, S. Mangrulkar, M. Sun, and B. Bossan, "Accelerate: Training and inference at scale made simple, efficient and adaptable." https://github.com/huggingface/accelerate, 2022.

[42] C. Wolf and J. Glaser, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[43] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh *et al.*, "Freepdk: An open-source variation-aware design kit," in *2007 IEEE international conference on Microelectronic Systems Education (MSE'07)*. IEEE, 2007, pp. 173–174.

[44] X. Bi, D. Chen, G. Chen, S. Chen, D. Dai, C. Deng, H. Ding, K. Dong, Q. Du, Z. Fu *et al.*, "Deepseek llm: Scaling open-source language models with longtermism," *arXiv preprint arXiv:2401.02954*, 2024.

[45] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano *et al.*, "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.

[46] M. Suzgun, N. Scales, N. Schärli, S. Gehrmann, Y. Tay, H. W. Chung, A. Chowdhery, Q. Le, E. H. Chi, D. Zhou *et al.*, "Challenging big-bench tasks and whether chain-of-thought can solve them," in *Findings of the Association for Computational Linguistics: ACL 2023*, 2023, pp. 13 003–13 051.

[47] Z. Yang, P. Qi, S. Zhang, Y. Bengio, W. Cohen, R. Salakhutdinov, and C. D. Manning, "Hotpotqa: A dataset for diverse, explainable multi-hop question answering," in *Proceedings of the 2018 conference on empirical methods in natural language processing*, 2018, pp. 2369–2380.