A Hybrid Processing-in-Memory Architecture for Long Sequence LLM Inference with KV Cache Filtering

PACT 2026 Submission **#109– Confidential Draft – Do NOT Distribute!!**

Abstract

Processing-in-Memory (PIM) is a promising solution for acceleration of attention operations in large language model (LLM) inference with its support for high internal bandwidth. However, with the growth of sequence lengths, the size of KV cache and the amount of operations required for attention linearly increase. To mitigate the capacity and computation problem of long-sequence LLM, dynamic KV cache filtering techniques have been proposed, which utilize only a few selected critical tokens for attention computation in each iteration. However, applying dynamic KV cache filtering schemes directly to existing PIM architectures may lead to unsatisfactory results: As it generates unstructured and dynamic sparsity in attention operations, it may cause hardware underutilization or redundant operations due to the all-bank control in conventional PIM architectures designed for dense attentions. In this paper, we propose Pascal, a hybrid PIM-DRAM architecture for sparse attention operations with dynamic KV cache filtering. Pascal contains three key designs: A novel data layout for load balancing with locality awareness, a hardware-based data divergence handling mechanism for efficient sparse operation, and an extension to PIM-DRAM hybrid systems with an efficient migration policy. Pascal outperforms the dense PIM by 5.3× speedup on average with PIM-only system, and improves the token generation throughput by 2.5× on average with PIM-DRAM hybrid system.

1 Introduction

The growing demand for long context understanding has been increasing sequence lengths generated by large language models (LLMs). The attention mechanism in decoder-based LLMs is both latency-critical and memory-intensive. (1) The operands—query (Q), key (K), and value (V)—are activation-derived and lack batched reuse opportunities, and (2) the size of the key-value (KV) cache grows linearly with the input length and batch size. To alleviate this bottleneck, previous studies have proposed offloading the attention kernel to processing-in-memory (PIM) architectures [7, 24].

Commercial PIM architectures provide high internal bank-level bandwidth for such memory-bound GEMV operations with *all-bank execution mode* [11, 13]. It allows commands to be shared efficiently across all the banks of the same channel to maximize bank-level parallelism. A command to a channel controls all the banks of the channel synchronously, and thus the same row or column address is used in each bank for a row activation or column access operation.

However, scaling PIM to support long-sequence inference poses two fundamental challenges. First, the KV cache becomes increasingly large, often exceeding the memory footprint of model weights. Second, the attention computation time increases rapidly with the sequence length due to the linear scaling of QK^T and PV operations. To address these issues, *KV cache filtering* selectively includes only *critical tokens*—those contributing significantly to attention output—across attention heads [4, 14, 30, 38]. This filtering approach reduces the size of the working set and the computational demand in attention computation, making long-sequence inference more tractable.

Despite its potential effectiveness, directly combining dynamic KV cache filtering with current all-bank PIM architectures introduces severe inefficiency. The conventional PIM architecture under all-bank execution is optimized for uniform and regular computation of dense GEMV computation. However, filtered attention induces unstructured token-level sparsity, where each head and layer selects different critical tokens. This mismatch gives rise to critical challenges: It can miss possible skipping opportunities. Under all-bank execution, any existence of critical token at a given row forces all banks to execute the corresponding operation - even if most banks could have skipped it. In addition, PIM capacity poses a limitation. While KV cache filtering reduces the amount of computation, it does not eliminate the need to store the full KV cache to preserve model accuracy. As the effective memory capacity of PIM is significantly reduced due to the on-die processing elements, using only PIM for storing the entire KV cache is not scalable for increasing KV cache sizes.

To address the challenges, we propose Pascal (PIM Architecture for Sparse KV CAche in Long-sequence LLM), a divergence-aware PIM architecture integrated with a PIM-DRAM hybrid memory system. The key insight is that all-bank execution should be preserved for efficiency, but augmented with mechanisms that support divergent row and column accesses under a unified control stream. Rather than abandoning synchronized execution, our divergence-aware PIM enables each bank to independently access different operands while maintaining the scalability benefits of all-bank execution. The divergence-aware PIM is backed by capacity-centric non-PIM memory, and critical tokens are migrated to PIM incrementally to minimize the disruption of the filtered attention to copy critical tokens to PIM.

Pascal addresses the compute, capacity, and control challenges through three co-designed components: First, we introduce a *filtering-aware* unified KV cache layout that partitions memory at the granularity of attention heads and aligns the memory layout with row-wise DRAM access. This design ensures load balancing across banks and enables row-local access patterns for both $QK^\top$ and PV computations, reducing row activation overhead.

Second, we propose a first novel architectural support for resolving *operand-level data divergence* for efficient sparse operations for all-bank execution. Our design allows each bank to access a different row or column from a shared command by generating row and column indices from the metadata stored in a reserved space of the bank. During inference, PEs dynamically resolve operand

PACT 2026, October 19–October 22, 2026, Chicago, Illinois, USA
2026. ACM ISBN 978-X-XXXX-XXXX-X/XX/XX
https://doi.org/XXXXXXX.XXXXXXX

addresses based on the stored indices while maintaining bank-level parallelism of all-bank execution.

Finally, to overcome the capacity limitation of PIM, we integrate the divergence-aware PIM architecture into a *PIM-DRAM hybrid system* for enhanced scalability. In the hybrid system, critical tokens must be migrated from the low-cost LPDDR memory to the PIM at each iteration. Our contribution is that we exploit the strong temporal locality in token reuse through a *partial migration* scheme that migrates a certain critical token only when it was not selected as a critical token in previous iterations, minimizing the amount of data to fetch from LPDDR.

We evaluate Pascal with a cycle-accurate simulator running state-of-the-art LLMs [8, 21, 32] with a dynamic KV cache filtering algorithm [3, 30] for PIM-only and PIM-DRAM hybrid scenarios. Our results show that for PIM-only evaluation, Pascal achieves 5.3× speedup over dense PIM baselines on average, and improves token generation throughput by 1.9× in an NPU-PIM combined system at a batch size of 4. Furthermore, the PIM-DRAM hybrid system with the proposed partial migration scheme improves the token generation throughput by 2.5× on average compared to dense baselines at a batch size of 8.

The main contributions are as follows:

- We identify the fundamental mismatch between dynamic KV cache filtering and all-bank PIM execution, showing the data divergence problem.
- We propose a filtering-aware KV cache layout with head-wise partitioning and row-local mapping, improving load balance and row-buffer locality.
- We introduce a divergence-aware PIM architecture that preserves all-bank execution while enabling per-bank operand selectivity, via metadata-driven row/column address translation without per-bank command overhead.
- We design a PIM–DRAM hybrid system that decouples bandwidth and capacity, using temporal-locality-aware partial migration to minimize KV cache movement.

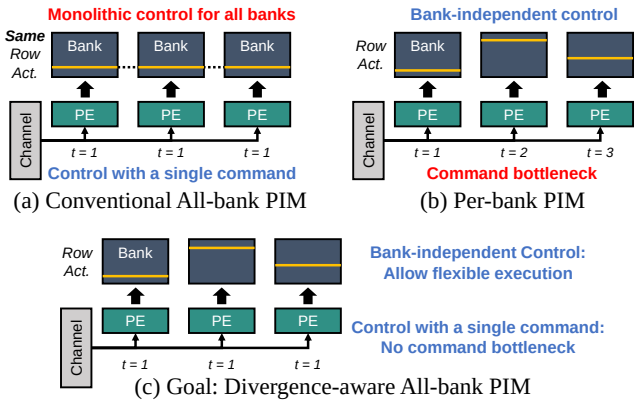
2 Background

2.1 Processing-in-Memory (PIM)

PIM enables efficient acceleration of memory-bound workloads by leveraging the high internal bandwidth of DRAM.

All-bank PIM Execution. DRAM-based PIM systems can orchestrate in-DRAM processing either independently within each bank (per-bank execution) or in lock-step across all banks in a channel (all-bank execution). To fully utilize bank-level parallelism, recent PIM architectures adopt the all-bank execution model shown in Figure 1 (a) [7, 11, 13, 24]. With a single broadcast command that triggers identical operations across every bank, all-bank execution can avoid serially sending commands to multiple banks in per-bank execution.

All-bank Divergence: Control and Data. Uniform control of all-bank execution, however, collides with the non-uniformity of sparse workloads. When only a subset of banks contains valid operands (*control divergence*), the monolithic control of all-bank execution might not ensure the correctness of the operation (i.e. may perform



(a) Conventional All-bank PIM (b) Per-bank PIM

(c) Goal: Divergence-aware All-bank PIM

Figure 1: Three execution modes of DRAM-based PIM: Per-bank, all-bank, and divergence-aware all-bank.

computation with invalid operands). Prior work proposed a conditional exit (CEXIT) technique to allow an inactive bank to bypass execution locally to mitigate control divergence, when the size of operand tensor differs across banks [2]. *Operand-level data divergence*, on the other hand, incurs another challenge: This is the case where every bank must compute, but the relevant operands reside at different row/column addresses. Because an all-bank execution allows only one shared address for each command, banks cannot simultaneously fetch operands at different positions. Data divergence is therefore a fundamental barrier that demands architectural mechanisms that allow selective access to the per-bank operand without sacrificing the unified control stream.

Necessity of All-bank Execution for Sparse Workloads. A naïve approach to eliminate both types of divergence is per-bank execution, depicted in Figure 1 (b). Each processing element receives its own command and may activate a distinct row in each associated bank, providing flexibility well suited to sparse and bank-specific access patterns [36]. However, the *command bottleneck* (i.e. bottleneck from excessive number of commands required for per-bank execution) throttles the throughput even within the sparse workloads [2]. Consequently, the research challenge is not to abandon all-bank execution but to embed divergence-tolerant features. Figure 1 (c) conceptually illustrates the all-bank PIM with data divergence handling to reconcile sparsity with synchronized control.

2.2 PIM for Large Language Models (LLMs)

Memory Bandwidth Challenges in LLMs. Transformer-based LLMs commonly suffer from memory bandwidth bottlenecks during inference on long sequences. LLM inference consists of prefill and decoding phase, with the latter being more memory-intensive due to repeated self-attention computations. The attention in the decoding phase is performed as an operation between the KV cache, which consists of all past keys and values, and the activations of the current input. Specifically, the main bottleneck in attention lies in the two **General Matrix-Vector multiplication (GEMV) operations** involving the KV cache, referred to as $QK^\top$ (Logit) and PV (Attend). This GEMV kernel lacks batching opportunities and requires frequent memory access, which makes the process bandwidth-bound. To alleviate bandwidth pressure, recent work

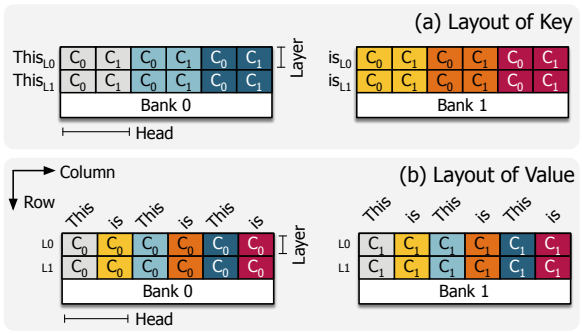


Figure 2: Subarray layout across DRAM banks for KV cache in PIM for dense LLM [7].

proposes all-bank PIM architectures that execute attention directly in memory [7, 24, 27, 28]. These designs reduce data movement and improve the efficiency of attention computation during decoding.

Acceleration of LLM with PIM. Researchers have proposed several PIM-integrated systems for LLM acceleration, combining PIM with other accelerators [6, 7, 12, 15, 18, 24]. PIM architectures accelerate memory-bound attention by distributing KV cache entries across DRAM banks and performing tiled GEMV operations in parallel [7, 24]. Each bank stores a portion of the KV cache, while query vectors are broadcast to all banks to enable synchronized computation. Given the low arithmetic intensity of KV access, offloading computation to PIM units reduces data movement and exploits high internal DRAM bandwidth.

KV Cache Layout. Prior work for LLM PIM implemented GEMV kernels by performing dot product between the DRAM row buffer (KV cache) and the PIM-side buffer (input vector), while ensuring independence between matrices stored in each bank [6, 7, 12, 24]. Although the K cache and V cache share the same topology (i.e. n $1 \times d$ entry vectors), they have different access pattern when used for the attention computation, as K cache should be transposed while V cache does not. Because of this, in conventional dense LLM PIMs, **the K cache and V cache are mapped onto the DRAM subarray in different ways**. Figure 2 describes a KV cache layout proposed by the prior work for dense LLM PIM [7]. Since K cache is multiplied by a query vector Q for 'a single (input) token', as shown in Figure 2 (a), each row of subarray should contain the whole entry vector (i.e. the row vector of K cache) for a single token, allocating entries to a bank in token granularity. V cache, on the other hand, is multiplied with an attention weight vector P for 'every token', thereby a row of each subarray should store a V cache element for one attention channel and all tokens, as illustrated in Figure 2 (b).

2.3 KV Cache Filtering for Long Sequence LLMs

The growing context length of LLMs imposes significant memory and latency burdens due to the proportional growth of the KV cache [1, 21, 23, 31, 34]. To mitigate this, KV cache filtering has emerged as an essential technique for long-sequence inference. Recent dynamic filtering methods offer a more robust solution [3, 4, 14, 17, 19, 30] than early static methods [38].

Dynamic KV cache filtering methods selectively retain only the most critical tokens by computing a **filtering score** for each token

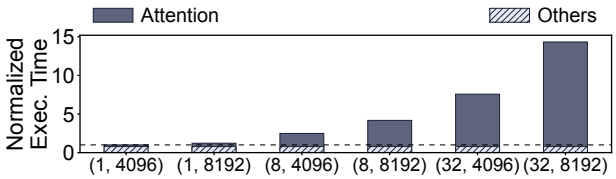


Figure 3: Execution time breakdown of Llama 3.1-8B decoder when attention is offloaded to PIM in NPU-PIM system for various batch sizes and sequence lengths.

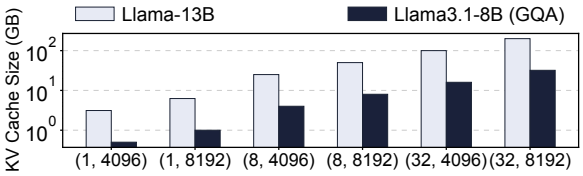


Figure 4: Growth of KV cache size with various batch sizes and sequence lengths, for LLMs in FP16 precision.

based on the current input query. This score approximates the token's actual attention contribution, and only the top-k tokens with the highest scores are used. This principle is realized through various algorithms approximating/predicting the attention score using a subset of KV cache [4, 14, 17, 19, 30]. Regardless of the specific algorithm, these dynamic methods yield a crucial characteristic for hardware design: **The set of critical tokens is highly dynamic**. That is, the distribution of critical tokens along the sequence varies by attention heads, layers, and decoding iterations. This dynamic and unstructured sparsity pattern motivates the need for flexible, context-aware hardware acceleration.

3 Motivation

3.1 Challenges in Long-sequence LLMs

Computation Constraints. To reduce the host-side load and improve inference throughput, prior PIM-based LLM systems offload memory-intensive attention GEMV kernels to near-memory compute units [7, 24]. This offloading significantly improves performance compared to executing all kernels on host NPUs or GPUs. However, the execution time of attention increases rapidly with longer sequence lengths and larger batch sizes due to the quadratic growth of the KV cache. Figure 3 presents the execution time breakdown of Llama 3.1-8B decoder, where attention is offloaded to HBM-based PIM [13] and remaining kernels run on a cloud-scale 8-core NPU [5]. The microbenchmarks are labeled with (batch size, sequence length) pairs, and execution times are normalized to the (1, 4096) case. The attention runtime contribution grows with sequence length and batch size, eventually becoming the dominant bottleneck even when offloaded to PIM [7].

Capacity Constraints. Prior systems adopt dense PIM layouts where each bank contains a PE and KV cache entries [6, 7, 24, 27]. However, this tightly couples compute with capacity and leads to inefficient provisioning for long sequence length. Figure 4 illustrates the rapid growth of KV cache size for various (batch size, sequence length) pairs. Even with grouped-query attention (GQA), which reduces KV size by grouping multiple heads, the total KV

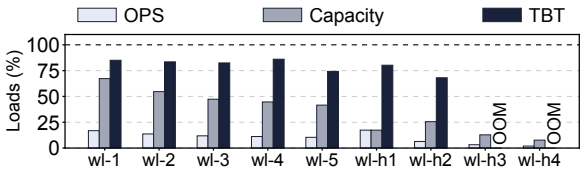


Figure 5: Impact of KV cache filtering on a GPU [30] in a view of arithmetic operations (OPS), memory footprint (Capacity), and latency (TBT), normalized to a dense baseline. Lower is better. OOM denotes an out-of-memory error.

cache size continues to grow linearly with the batch size and sequence length. Under long contexts or large batches, the KV cache becomes dominant in memory footprint and can exceed the size of model weights. This trend creates a critical trade-off in PIM-based systems. For PIM architectures where every bank contains PEs, increasing memory capacity by adding more banks may lead to PE underutilization, while reducing banks limits capacity.

Solution: KV Cache Filtering. To address the challenges in long-sequence LLMs, dynamic KV cache filtering has emerged as a key solution [4, 14, 17, 19, 30]. By selecting only critical tokens for attention, filtering significantly reduces the amount of data and computation. However, applying filtering directly to conventional systems results in suboptimal performance.

Limitations of GPU-based System. Although dynamic KV cache filtering reduces computational workload and memory footprint, the achievable speedup remains bounded by GPU architectural constraints. Figure 5 illustrates this constraint on an NVIDIA H100 with 80 GB of memory running the open-source implementation of dynamic KV cache filtering [30] and the workloads in Table 3. Filtering reduces the attention layer's operation count (`OPS`) by between 83% and 96.8% of the dense baseline, and shrinks the cache footprint (`Capacity`) by between 32.3% and 92.3%. Note that the dense baseline already reflects the footprint reduction in GQA, resulting in the gap between `OPS` and `Capacity`. Despite these reductions, the maximum reduction of time-between-token latency (`TBT`) is only 32%. This is because GPU is designed for compute-intensive workloads with high levels of parallelism and massive data reuse with regular access patterns, which is insufficient for memory-intensive sparse attention that incurs a memory bandwidth bottleneck with irregular access pattern. Specifically, Quest exploits sparsity by loading only KV cache entries of critical tokens from global memory to shared memory and performing the dot product between input activation and KV cache entries (i.e., K vector and V vector) in each thread. This reduces the throughput efficiency of GPUs, as the GPU is required to execute kernels with significantly lower arithmetic intensity than it was originally designed for. This result underscores the need for an architecture for memory-bound sparse kernels, emphasizing the necessity of PIM-based acceleration.

3.2 Limitations of Dense PIM with KV Cache Filtering

Several prior works have proposed PIM-xPU systems offloading attention operations to PIM to efficiently accelerate memory-intensive attention kernels [6, 7, 12, 15, 24]. However, despite their advantages

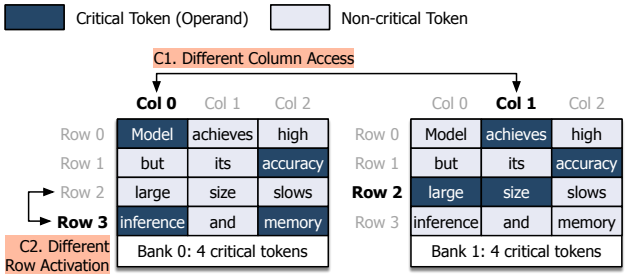


Figure 6: Challenges of dense layout and naive all-bank PIMs for sparse attention with KV cache filtering. Note that a word on each token is only an example for better understanding, not indicating the real mapping.

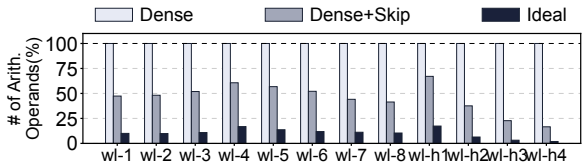


Figure 7: Relative number of arithmetic operations for dynamic skipping, compared to the dense baseline and ideal case (perfect skipping) in $QK^\top$ kernel.

on a memory bandwidth, conventional dense PIMs are architecturally incompatible with the **unstructured sparsity** from KV cache filtering with **long sequence length**. KV cache filtering introduces nonuniform distribution patterns of critical tokens across heads [3, 4, 14, 30]. This makes the physical locations of critical tokens vary due to head- and layer-specific filtering decisions, causing two major challenges described in Figure 6. Moreover, because the set of tokens selected as critical tokens in dynamic KV cache filtering can vary at each iteration, even non-critical KV cache entries must be stored somewhere, which introduces another challenge.

Challenge 1. Data Divergence. In all-bank PIM, the same command is executed simultaneously across all banks and rows within a channel. This incurs *data divergence* in sparse workloads. Figure 6 shows the two types of *data divergence*. **C1** indicates *column-access divergence*, making the operand of each bank row reside in different columns (Col 0 for Bank 0, and Col 1 for Bank 1). **C2** shows *row-access divergence*, where each bank should **activate the different row** for the same logical operation (Row 2 for Bank 0, and Row 3 for Bank 1). Such divergent access patterns are not considered in conventional all-bank PIMs.

Although reading the row/column without operands and skipping the computation with *dynamic (conditional) skip* would allow the all-bank execution, this approach still diminishes the advantage of KV cache filtering. Figure 7 quantifies this impact with the workloads of Table 3 by counting the number of arithmetic operations for $QK^\top$ (Logit) kernel. `Dense` represents the baseline without filtering, while `Dense+Skip` is the PIM with *data divergence* including column-access divergence caused by KV cache filtering. `Ideal` assumes ideal filtering with exactly same distribution of critical tokens in each bank. Although `Dense+Skip` reduces arithmetic operations by up to 6.0× compared to `Dense`, it still requires up to 8.7× more operations than `Ideal`, resulting in performance degradation.

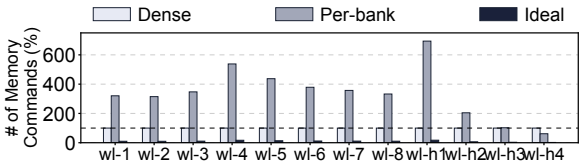


Figure 8: Relative number of per-bank memory commands, compared to the dense baseline and ideal all-bank (no divergence).

Another alternative solution is to issue per-bank commands for data load (Figure 1 (b)), but this dramatically increases command traffic and worsens latency. Figure 8 compares the number of memory commands (i.e. bank-to-PE load) required for execution with real-world workloads of Table 3 under three modes: `Dense`, `Per-bank`, and `Ideal`. Per-bank mode requires up to 40× more and 8.6× more commands than the ideal case and baseline, respectively. This would introduce significant performance degradation attributable to the *command bottleneck*. These results suggest that neither per-bank operation nor conditional execution is sufficient to handle data divergence.

Challenge 2. System Capacity Constraints. While dynamic KV cache filtering reduces the compute demand of long-sequence inference, the underlying KV cache must still be stored in memory. However, provisioning the entire memory capacity with PIM banks alone can be impractical due to system or user constraints. For instance, an on-device system may face area and cost limitations when integrating more PIM banks. Moreover, since PIM must accommodate both banks and PEs within a limited die area, the presence of PEs can reduce the size of bank subarrays, thereby decreasing the overall memory capacity of the PIM device itself [2, 13]. For example, in Samsung HBM-PIM, the die with PIM support provides only half the capacity compared to a conventional HBM2 die [13]. As batch sizes increase, the required capacity for KV cache size scales accordingly, quickly exceeding the capacity that can be provided by PIM even with KV cache filtering.

Limitation of Prior Work. Although there exist prior works addressing related problems, they do not provide a complete solution for accelerating PIM-based dynamic KV cache filtering due to the differences in the problem scope. PIMphony introduces a hardware-software co-design to mitigate the attention acceleration performance degradation with long context without filtering. However, its scalability is limited with the rapid growth of context lengths [10]. pSyncPIM is an all-bank PIM architecture designed for sparse acceleration, but it fails to resolve the data divergence (**C1**, **C2**) since it assumes the compressed format for sparse data [2]. While LongSight applies dynamic KV cache filtering to GPU-PIM system, it maps filtering operation to PIM and sparse attention to GPU [26]. As a result, it still inherits the inefficiency of GPU (Figure 5) while also requiring the excessive PIM memory capacity.

4 Data Divergence-aware PIM Design

4.1 KV Cache Layout

Figure 9 illustrates the proposed filtering-aware layout for the key-value (KV) cache, along with its hardware mapping to *PV* (Attend)

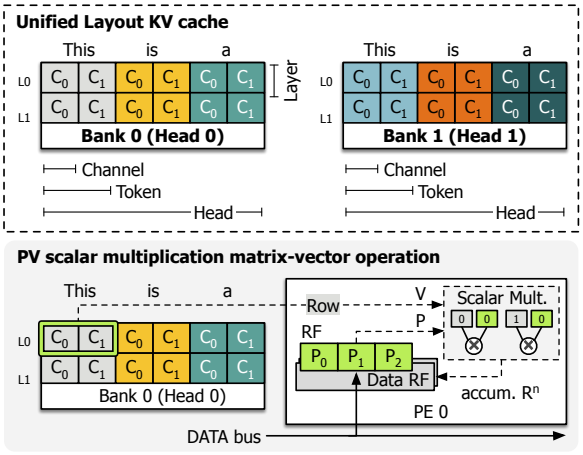


Figure 9: Filtering-aware KV cache layout using head-wise partitioning and row-wise access, with an example of *PV* execution with Algorithm 1.

Algorithm 1 Scalar multiplication-based algorithm of *PV*.

```
P: 1 × n vector, V: n × H matrix, S: 1 × H vector
for i = 0 to n do
    for j = 0 to H do
        S[j] = S[j] + (P[i] × V[i][j])
    end for
end for
```

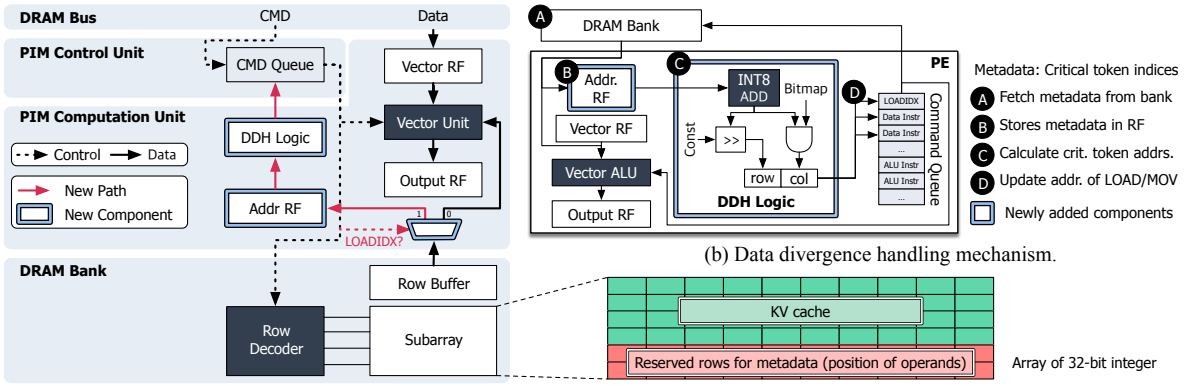
operation. The layout is based on two design principles: (1) attention head-level partitioning, and (2) an identical mapping strategy for keys and values. With *attention head-level partitioning*, each PE (bank) is mapped to the KV cache of a single attention head. Therefore, KV cache vectors from different heads are mapped to different PEs (banks). Due to the independence between attention heads, unlike the dense layout of Figure 2, this partitioning scheme does not require any data sharing between PEs. *Identical mapping strategy* means that K cache and V cache share the same data-to-subarray mapping scheme as follows:

(a) Entry-level mapping: As shown in Figure 9, the key/value vectors are sorted in row-wise manner in subarray. Each element of the vector is ordered by the KV cache channel index, and each token is ordered by the token index. The key difference between the prior dense K cache layout (Figure 2 (a)) and our filtering-aware layout lies in the head-level partitioning.

(b) Layer-level mapping: KV cache entries with the same head index but from different layers are mapped to the same PE (bank), ordered by the layer index.

The proposed layout offers several benefits:

Improving Data Locality via Algorithmic Reformulation. To fully leverage the proposed row-wise KV cache layout, the computation must exhibit frequent reuse of activated rows. However, standard dot-product-based GEMV algorithms used in the *PV* operation access the value cache in a column-wise fashion, resulting in low row buffer locality and frequent row activations. To address this inefficiency, we reformulate the *PV* computation using *scalar multiplication*, which accesses value vectors in a row-wise manner. Algorithm 1 outlines the computation of *PV* using this reformulated



(a) PIM Workflow. DDH indicates Data Divergence Handler.

(b) Data divergence handling mechanism.

(c) DRAM bank subarray configuration.

Figure 10: Data divergence-aware PIM architecture.

Type	Instruction	Operation
Data	LOADIDX addr, size	Load metadata in $Mem[addr]$ of size $size$ to the Addr RF
	LOAD addr, size	Load data in $Mem[addr]$ of size $size$ to $RBuf$ (Row activation)
	MOV addr, size	Move data in $RBuf[addr]$ of size $size$ to ALU RF

Table 1: Data instructions of data divergence-aware PIM. RF, *Mem*, and *RBuf* indicates PE register file, DRAM bank, and DRAM row buffer, respectively.

approach. Figures 9 (a) and (b) demonstrate how each operation is mapped to the proposed KV cache layout. Since both dot product and scalar multiplication can be efficiently implemented using vector array units [9, 13], the proposed reformulation can be adopted without complex hardware modifications.

Efficient Insertion of New KV Entries: The proposed layout stores each vector row-wise, limiting the number of activated rows and improving write efficiency. Furthermore, keys and values share the same layout, eliminating the need for format conversion and reducing the associated overhead.

4.2 PE Architecture and Execution Flow

Instruction Set Architecture. We propose a custom ISA derived from the ISA of prior work [13], modified for sparse attention with KV cache filtering. The proposed ISA contains three types of instructions with distinct formats: Control, data, and vALU (vector ALU). It shares the same set and format of control instructions (`NOP`, `JUMP`, `EXIT`) with the baseline ISA, and extends the scalar ALU instructions to vALU instruction: `VMAC` (extension of `MAC`) and `REDUCE` (extension of `ADD`) are for implementation of dot product-based GEMV, and `VMAD` (extension of `MAD`) is an instruction for scalar multiplication. Table 1 summarizes the information about data instructions. `LOADIDX` is a data instruction for fetching critical token index metadata from a DRAM bank for *data divergence handling mechanism* (Figure 10 (b)), which is one of the key designs of the proposed PIM design and will be explained in Section 4.3. `LOAD` and `MOV`, on the other hand, is a data instruction that precedes vALU instructions to load the operand (i.e. critical KV cache entry vector) from a subarray to an ALU register file (ALU RF).

PE Design and Execution Flow. Figure 10 (a) shows the overall architecture and workflow of the processing element (PE) of the data divergence-aware PIM and its interaction with the DRAM bank. As shown on the left side of Figure 10 (a), the PE includes a command queue to buffer incoming PIM commands (instructions) from DRAM bus in FIFO order. Instructions stored in the command queue are executed in order, to trigger activate/read operation on DRAM bank and/or to make vector unit perform computation. When a `LOADIDX` instruction is detected, the data retrieved from the DRAM bank is metadata (i.e. critical token indices) and must be utilized by the data divergence handler logic (DDH Logic), while this data should be forwarded to the vector unit for any other command. This decision is made by a 1-to-2 demultiplexer connected to the row buffer. The demultiplexer receives a 1-bit control signal which is set to 1 only when `LOADIDX` instruction is detected. A value of 1 routes the data to the address register file (Addr RF), whereas a value of 0 directs it to the vector unit. The metadata loaded by `LOADIDX` instruction is stored in Addr RF to be utilized in data divergence handling mechanism in further execution, which will be explained in detail in the following subsection. For computation, `LOAD` instruction activates the specific row, MOV instructions fetch specific columns from the row buffer into the ALU RF in vector unit, and vALU instructions trigger vectorized arithmetic operations between the data loaded in ALU RF and vector register file (Vector RF).

Support for Dense Attention. The proposed PIM architecture supports both sparse and dense attention [30]. By extending a *mode register* concept of prior work [13], the data divergence-aware PIM uses an 1-bit *Execution Mode Register* to distinguish the sparse and the dense mode. If the value of execution mode register is 1, sparse attention is performed. Otherwise, dense attention is performed by bypassing components and paths marked blue in Figure 10 (a).

4.3 Handling Data Divergence

Critical Token Data Divergence. As the distribution of actual critical tokens differs across heads and decoding iterations, each PE must access the different position of a bank subarray to fetch the correct critical tokens. This creates an *operand-level data divergence*, where the same instruction must access different addresses across banks. As discussed in Section 3.2, resolving this divergence through per-bank memory command leads to excessive command traffic.

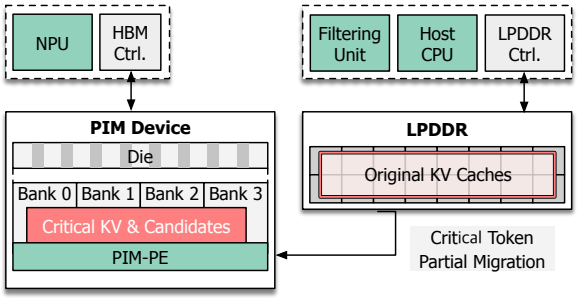


Figure 11: Overview of PASCAL, a PIM-DRAM hybrid system for KV cache filtering.

Critical Token Indexing via Metadata. Each DRAM bank reserves a metadata region that holds the critical token indices (i.e. position of critical tokens among the whole context sequence) to be used in the current decoding iteration. This region, highlighted in red in Figure 10 (c), enables per-bank operand selection without requiring divergent instructions. As metadata encodes the position of critical tokens among the context (i.e. sequence of tokens), 32-bit integer is a proper precision for it. The total size of metadata per one head and one decoder is $n \times 4$ bytes, where n indicates the number of critical tokens per head. For benchmarks of Table 3, the ratio of the metadata size to the total KV cache size is only 0.23% on average. This metadata is generated in the host-side filtering unit, considering that their light-weight and specialized operations that are not suitable for PIMs [4, 14, 30]. As the size of critical token metadata is much smaller than original KV cache and metadata migration is required only once for each decoder layer and head (PE), the time required for writing metadata in per-bank manner is about hundreds of nanoseconds, which is less than a few percentages of the time required for attention computation.

Metadata Reference-based Divergence Resolution. We introduce a hardware-assisted mechanism to resolve data divergence in sparse attention by referencing the metadata stored in the reserved area of banks. Basically, the proposed mechanism updates the address of memory access commands (i.e. destination of `LOAD`) in command queue based on the *critical token indices metadata* in the bank We enable this mechanism in PIM through a hardware-based *data divergence handler* (DDH), shown in Figure 10 (b).

The following steps describe the execution flow with data divergence handling mechanism: Ⓐ After KV cache filtering, the critical token indices of the current layer are updated and stored to the reserved area of the bank (Figure 10 (c)). Ⓑ Before the attention computation is initiated, the critical token indices metadata is fetched to the Address Register File (Addr RF). Since K cache and V cache shares the same layout of Section 4.1, metadata fetching process (Ⓐ) is performed only once per decoder layer for each PE, and the temporal overhead required for this metadata access is less than 1 microsecond. Ⓒ After the attention computation is initiated, *Data divergence handler* logic calculates the subarray row/column index (i.e. absolute position on a bank) of critical tokens. Ⓓ The source address of Data instructions in the command queue is updated with the absolute address simultaneously with the sparse attention computation. Each PE performs this process independently, as the filtering-aware layout avoids inter-PE data dependencies. By generating the source address internally from critical token position metadata, this mechanism enables each bank to independently determine the memory addresses of critical tokens while maintaining all-bank instruction synchronization.

Intra-PE Synchronization. The proposed PIM architecture contains an additional path in sparse mode. To ensure the synchronized execution, the latency of DDH Logic should be shorter than the clock cycle of PE (i.e. 4 ns for 250MHz frequency) to update the destination of commands in command queue in time. Considering the lightweight components inside the DDH logic [22, 25], passing through this logic does not compromise synchronization.

5 PASCAL: PIM-DRAM Hybrid System for KV Cache Filtering

Based on the data divergence-aware PIM design of Section 4, we propose PASCAL, a **data divergence-aware PIM-DRAM hybrid architecture with memory heterogeneity**. As visualized in Figure 11, the low-tier LPDDR offers high memory capacity at low cost and power, serving as the main storage for the original KV cache. Conversely, HBM banks of data divergence-aware PIM (i.e., PIM banks) are reserved for critical tokens of current and previous iterations, as outlined in Section 5.2. As shown at the top of Figure 11, the host accelerator integrates both a general-purpose NPU for GEMM kernels in the decoder and a Filtering Unit for dynamic KV cache filtering. This modular separation between compute and memory resources allows the system to scale for LLM workloads with even longer sequence lengths, achieving considerable performance, cost, and energy efficiency.

5.1 PIM Die Configuration

Balancing Compute and Capacity. Attaching a dedicated PE to each PIM bank leads to a high compute-to-capacity ratio, reducing memory efficiency. To address this, we adopt a shared-PE architecture, where **a single PE is connected to multiple banks within a PIM die**, following prior work [13]. The distance from each PIM bank to the associated PE is equivalent. Therefore, from the perspective of the PE, this configuration is effectively equivalent to connecting to a single large memory bank. This approach maintains the computing power of PIM while enabling flexible capacity scaling at reduced area and power costs [28].

Layout Extension. Extending the layout of Section 4.1, each attention head is mapped to a single PE instead of a single bank, and the KV cache is distributed across banks connected to the PE with row-wise interleaving. This extended layout efficiently utilizes the internal bandwidth while preserving the access locality.

Compatibility with GQA. Although GQA reduces overall KV cache size, critical token sets remain distinct across heads even within the same group [30]. Thus, critical tokens must be managed and mapped at the granularity of heads. While dynamic KV cache filtering removes the effect of GQA, it still provides capacity benefits depending on the ratio of critical tokens, as shown in Figure 5.

5.2 Data Migration between Memory Modules

Migration Overhead in Hybrid Memory. In a proposed PIM-DRAM hybrid system, the original KV cache is stored in low-tier LPDDR, whereas computation with the critical KV cache must be

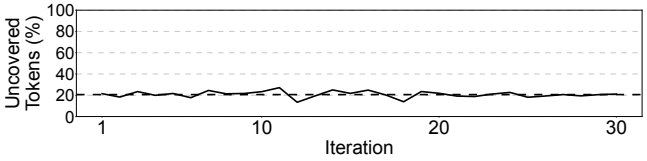


Figure 12: Average ratio of critical tokens that have not been selected as critical tokens in previous iteration (denoted as 'Uncovered Tokens'). Dotted line indicates the average.

performed in PIM. This necessitates migrating the critical portion of the KV cache to PIM at each decoding iteration. Simply transferring all critical tokens or the entire KV cache from LPDDR to PIM banks for every decoder layer results in excessive migration, incurring substantial latency and bandwidth overheads due to repeated memory controller interventions. To mitigate this, we design a migration strategy that reduces unnecessary data movement.

Temporal Locality of Critical Tokens. According to recent studies, adjacent iterations in the decoding phase exhibit a high likelihood of having similar attention score distributions and values [33, 37]. According to the analysis in prior work, the average similarity of queries between adjacent decoding iterations exceeds 80%, which implies the high similarity of attention scores [37]. In dynamic KV cache filtering, critical tokens are selected based on approximated attention scores. Consequently, such inter-iteration similarity naturally induces temporal locality for critical tokens. Figure 12 visualizes the temporal reuse patterns of critical tokens during 30 decoding iterations for a real-world workload (`wl-4` of Table 3), illustrating the degree of locality. The black line presents the ratio of 'Uncovered Tokens' for each head in each sparse decoder layer, which is defined as critical tokens in the current iteration that are not found in the previous iteration's critical token set. The x-axis indicates the number of cumulative iterations, the y-axis indicates the ratio of uncovered tokens, and the dotted line represents the average. The average ratio of uncovered tokens across all sparse decoder layers is 20.6%, implying that 80% of the critical tokens are reused between adjacent iterations. This result confirms that critical token selection exhibits strong temporal locality.

Partial Migration. Based on the previous analysis, we adopt *partial migration* as a migration policy for the critical tokens in Pascal. This policy assumes realistic PIM device capacity constraints where more tokens than critical tokens can be stored, but not enough to accommodate the entire KV cache of long contexts. In the first decoding iteration, all critical tokens are migrated from LPDDR to PIM device. In subsequent decoding iterations, attention computation is performed on tokens already buffered in PIM banks, and additional tokens are brought in as needed through controlled data migration. This policy leverages temporal locality to maintain minimal migration cost. Specifically, this scheme involves two types of events during migration:

(1) On-demand Migration: After the set of critical tokens is determined, only those tokens not currently present in the PIM device are fetched from LPDDR. The average migration volume is only 4.2MB in the sample of Figure 12.

(2) Victim Selection: If there is no available space to accommodate a newly fetched critical token, one of the existing tokens is selected as a victim and overwritten with the new token. Since PIM banks are include DRAM, writebacks are unnecessary. We employ a first-in-first-out (FIFO) victim selection policy. As the distance between iterations increases, the similarity of attention scores decreases. Because of this, FIFO yields an effect similar to LRU or attention score–based policies while maintaining low overhead.

Combination with Data Divergence-aware PIM Design. As long as the tokens in the K cache and the V cache are aligned in the same order, the token ordering does not affect computational correctness. Therefore, there is no need to reorder newly added tokens; instead, data divergence still appears. This underscores the necessity of the data divergence-aware PIM design of Section 4.

5.3 Scalability of Pascal

When the total number of heads ($B \times N$) exceeds the number of available PEs, multiple heads are assigned to a single PE for sequential processing. In contrast, when $B \times N$ is smaller than the number of PEs or a head's footprint exceeds a PE's capacity (i.e., the sum of the capacities of banks associated with one PE), the KV cache requires finer-grained partitioning. Both the K cache and V cache are partitioned at the attention-channel granularity. After computing $QK^\top$, data must be transferred to the host to perform the softmax operation; the partial output vectors are then aggregated at that stage. Hence, partial-sum reduction does not require an inter-bank communication. As long as the vector unit is fully utilized (i.e., up to 8 PEs per head in our evaluation), the additional partitioning of the head does not incur performance degradation. Similar to the PE scenario, when the model is too large for all critical tokens to fit within a single Pascal device, multiple devices may need to be utilized. In this case, tensor parallelism is first employed to distribute the model as in prior work [7].

6 Evaluation

6.1 Methodology

Simulation. We develop a cycle-accurate simulator in C++ for modeling the timing behavior of DRAM-PIM integrated systems, based on DRAMsim3 and the Samsung HBM-PIM simulator [13, 16]. We model the host-side NPU following the configuration of a commercial NPU [5]. The detailed hardware configuration of our evaluated system is summarized in Table 2. The filtering overhead is also modeled for every sparsity-aware PIM configuration.

Workloads. We collect 12 real-world LLM workloads by applying dynamic KV cache filtering schemes [3, 30] to dense LLMs [8, 21, 32], summarized in Table 3. The workload suite covers 30 decoding iterations for a given prompt length. There are two groups in our workload suite: General workloads (`wl-1` to `wl-8`) and hybrid system workloads (`wl-h1` to `wl-h4`), distinguished by the KV cache volume and prepared for two types of scenarios.

Evaluation Scenario. We consider two scenarios for the evaluation of Pascal: PIM-only and PIM–DRAM hybrid system configurations. The PIM-only scenario assumes that the entire KV cache can be accommodated within PIM without LPDDR, in order to evaluate the effectiveness of the PIM design in Section 4. For this purpose, we use a batch size of 4 and a general workload set. The PIM–DRAM hybrid scenario increases the batch size to 8 and uses both hybrid system and general workload sets, corresponding to a capacity where the

PIM Device	
DRAM Timing Parameters	HBM2 (x128) Default Timing [16]
Device Width	128
# of PIM Dies	8
# of Channels per Die	8
Subarray Configuration	32768 rows, 32 cols
# of Banks/PEs per Channel	16/4
Compute Units	FP16 vector ADD, vector MUL
Vector Unit Width	32
Total Register File Size	16KB per PE
PE Frequency	250MHz [13]
System	
NPU Configuration	8-core, 128×128 PE, 16GB HBM2 [5, 7]
Filtering Unit Configuration	vADD, vMUL, vCMP (width 256)
Filtering Unit SPM Capacity	16MB
Host Accelerator Frequency	1GHz
PIM-only System Memory	**PIM 16GB**
PIM-DRAM Hybrid Memory	**PIM 16GB, LPDDR4 64GB (45GB/s)**

Table 2: Hardware configuration of PASCAL.

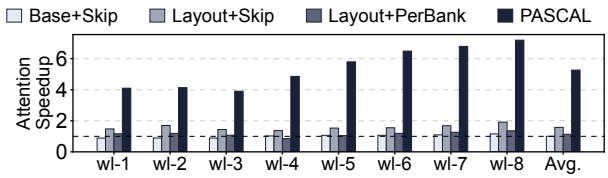


Figure 13: PIM-only system's sparse attention speedup normalized to the baseline, with a batch size of 4.

PIM device can store up to 4K tokens per head. We assume that all requests in a batch have the same sequence length.

Baseline PIM Design. We use the all-bank dense LLM PIM design as an evaluation baseline, borrowing the hardware architecture and KV cache layout of prior work [7], with an equal hardware budget with Table 2 for fair comparison.

6.2 Performance of PIM-only System.

Design Variants. We compare the performance of PASCAL against a baseline and three PIM variants that incorporate KV filtering under different layout and sparsity handling strategies: (1) Dense layout with *dynamic skipping* (`Base+Skip`), where computation is conditionally skipped for non-critical tokens; (2) Filtering-aware layout with dynamic skipping (`Layout+Skip`); and (3) Filtering-aware layout with per-bank read of critical tokens (`Layout+PerBank`) that resolves data divergence by replacing all-bank data instructions to per-bank instructions (Figure 1 (b)). The proposed PIM design (PASCAL) supports both the filtering-aware layout and fully-synchronized execution through a data divergence handling mechanism. Figure 13 shows the speedup of sparse attention with PIM designs normalized to the baseline. PASCAL achieves 5.3× speedup with sparse attention, notably outperforming other designs.

Effect of Data Layout. In Figure 13, the gap between `Base+Skip` and `Layout+Skip` shows the performance impact of KV cache layout. The speedup of `Layout+Skip` over `Base+Skip` is primarily due to an auxiliary effect: Improved load balancing between PEs. As a result, despite both designs employing dynamic skipping for sparse computation, `Layout+Skip` achieves a 1.5× speedup over `Base+Skip` on average.

Effect of Data Divergence Handling Mechanism. While `Layout+Skip`, `Layout+PerBank`, and PASCAL share the filtering-aware KV cache layout, they differ in how operand addresses are resolved during sparse attention execution. `Layout+PerBank` sends data instruction commands in per-bank manner to access token operands independently at each bank, incurring a *command bottleneck* as discussed in Section 2.1. As a result, its performance reaches only 0.22× of PASCAL. Next, `Layout+Skip` shows a 1.4× speedup over `Layout+PerBank` by using all-bank data instruction commands while skipping computation in banks that do not contain critical tokens. However, under the all-bank execution model, if even a single bank requires computation for a given operation, all banks must participate—limiting the skipping opportunity. This limits `Layout+Skip` to reach only 0.3× of the performance of PASCAL. In contrast, PASCAL allows each bank to resolve operand addresses independently based on local metadata, while still maintaining synchronized control. This enables selective operand access without inducing control divergence or excessive command overhead, allowing PASCAL to fully exploit both layout locality and filtering-induced sparsity for maximal throughput, achieving a 5.3× performance improvement over the baseline.

Name	Model	Task (Dataset)	Prompt Length	# Crit. Tokens	Filtering Scheme
wl-1	Mistral-7B	ruler	7,668	768	RocketKV [3]
wl-2			7,806	768	
wl-3			9,436	1,024	
wl-4	Llama3.1-8B	longbench	6,091	1,024	Quest [30]
wl-5			7,493	1,024	
wl-6			8,654	1,024	
wl-7			9,173	1,024	
wl-8			9,849	1,024	
wl-h1	Llama2-13B	passkey	11,421	2,048	
wl-h2	Llama3.1-8B		32,061	2,048	
wl-h3			64,061	2,048	
wl-h4			106,728	2,048	

Table 3: Real-world LLM workloads with dynamic KV cache filtering. 'h' denotes the large workloads added for PIM-DRAM hybrid system evaluation.

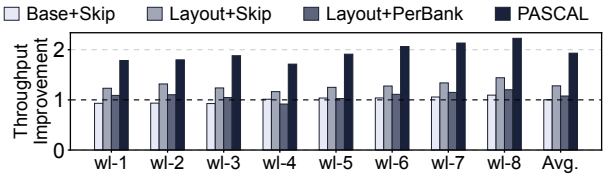


Figure 14: NPU-PIM system's token generation throughput improvement over the baseline, with a batch size of 4.

Token Generation Throughput of NPU-PIM System. Figure 14 presents the token generation throughput improvement of the NPU-PIM integrated system normalized to the baseline NPU-dense PIM integrated system. PASCAL achieves a 1.9× throughput improvement on average over the baseline when integrated with NPU. The smaller performance gain compared to the sparse attention-only evaluation (Figure 13) is due to the execution time of dense kernels: The performance of GEMM kernels in the decoder is independent of the sparsity of the KV cache. In addition, among the filtering schemes used in our evaluation, Quest does not perform filtering in the first and last decoder layers [30], which further reduces the performance improvement. Nevertheless, the result in Figure 14 shows that PASCAL is an effective solution for long-sequence LLMs.

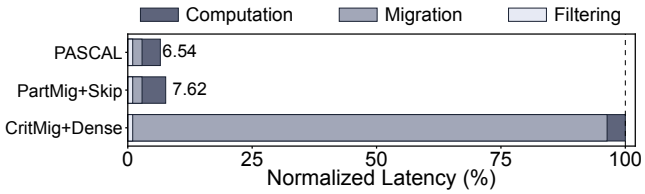


Figure 15: Latency breakdown of sparse attention in PASCAL with a batch size of 8, for the `wl-4` as an example.

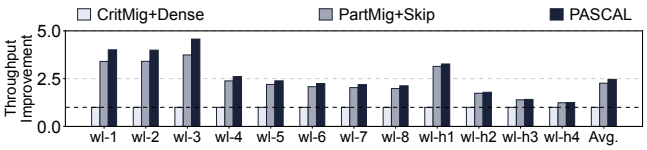


Figure 16: PIM-DRAM hybrid system's token generation throughput improvement with NPU integration, with a batch size of 8.

6.3 Performance of PIM-DRAM Hybrid System

Baseline System. Since a dense hybrid PIM–DRAM system without any filtering suffers from prohibitively low performance, we introduce a new baseline and control group to fairly evaluate the performance of PASCAL. (1) `CritMig+Dense`: This configuration employs a dense PIM design. At each iteration, all critical tokens are migrated, and dense attention is performed using the resulting compressed set of critical tokens. We use this as the baseline (i.e., performance normalization point). (2) `PartMig+Skip`: This configuration employs a dynamic skip-based PIM design (`Layout+Skip`) with a partial migration of PASCAL.

Effect of Migration Policy. Figure 15 presents a performance breakdown of sparse attention execution of PASCAL with a batch size of 8, for `wl-4` workload as an example. The x-axis of Figure 15 denotes the execution time normalized to the baseline (`CritMig+Dense`). Each bar is divided into three segments: the lightest color indicates the filtering overhead, the medium shade denotes the cost of KV cache migration from DRAM to PIM, and the darkest segment represents attention computation time. As shown in the figure, the overall performance is primarily determined by migration, since the low bandwidth of LPDDR acts as a bottleneck. In contrast, filtering is efficiently accelerated by the filtering unit, and therefore accounts for only a small fraction of the overall execution time. Therefore, `CritMig+Dense` suffers from the overhead of whole critical KV cache migration. On the other hands, `PartMig+Skip` and `PASCAL` significantly reduces migration time by transferring only the subset of critical tokens through partial migration, reducing 98% of temporal overhead from migration. Finally, the PIM design PASCAL provides 1.17× additional performance improvement over `PartMig+Skip` through efficient data divergence handling.

Token Generation Throughput. Figure 16 shows the token generation throughput improvement of the PIM-DRAM-NPU hybrid system, normalized to the baseline `CritMig+Dense`. `PartMig+Skip` reports a 2.3× throughput improvement on average by reducing the migration volume. Furthermore, PASCAL achieves a 2.5× throughput improvement on average over the baseline by combining temporal locality-aware migration and efficient data divergence handling.

	HBM-PIM [13]	**pSyncPIM** [2]	**PASCAL**
Area per PE (mm^2)	0.712	0.967 (+35.8%)	0.767 (+7.7%)

Table 4: Area overhead of PEs in PASCAL and prior work.

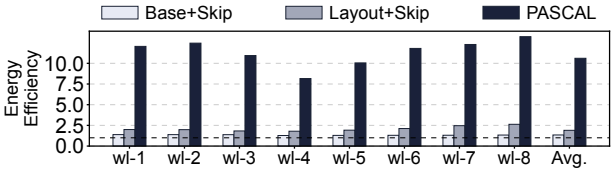


Figure 17: Relative energy efficiency of sparse PIM designs over the baseline, with a batch size of 4.

The lower performance gain compared to Figure 15 is due to the impact of the NPU and the presence of a non-sparse decoder in certain KV cache filtering schemes, similar to Figure 14. Moreover, since dense decoder layers require migrating all tokens for computation in the hybrid system, their execution time is substantially increased due to migration overhead and makes the performance gap among the three designs gets smaller. Despite these unfavorable conditions, PASCAL still demonstrates substantially superior performance over other designs through partial migration and data divergence-aware PIM design.

6.4 Area and Energy Efficiency

Area Analysis. We estimate the area overhead of PIM PE in PASCAL assuming 20 nm technology as prior work did [2, 13]. We collect the area of computation units and the command queue from [13], and the area of a 16KB register file for address RF is collected from [35]. Table 4 presents the area of the PASCAL PE in comparison with two HBM2-based all-bank PIM architectures with publicly available area data [2, 13]. PASCAL requires negligible additional area overhead to support sparse acceleration: 1.08× of HBM-PIM and 0.80× of pSyncPIM. This is because, unlike pSyncPIM targeting sparse HPC workloads, PASCAL does not contain complicated logic for metadata handling and data processing.

Dynamic Energy Efficiency. We measure the dynamic energy consumption of the PIM architecture in PASCAL by combining the DRAM row activation energy and the arithmetic operation energy [20, 29]. Figure 17 shows the energy efficiency of the data divergence-aware PIM design of PASCAL for sparse attention acceleration, represented as the inverse of total dynamic energy consumption normalized to the baseline. On average, `Layout+DataDiver` achieves 10.7× higher energy efficiency than the baseline, 7.9× over `Base+Skip`, and 5.6× over `Layout+Skip`. The result demonstrates that PASCAL is suitable for both high-throughput data center and energy-constrained on-device environments.

7 Conclusion

This paper proposes PASCAL, a PIM-DRAM hybrid architecture for sparse attention with KV cache filtering in long-sequence LLM. The PIM design of PASCAL achieves 5.3× speedup for the sparse attention, and PIM-DRAM hybrid system achieves 2.5× token generation throughput improvement over the baseline.

References

[1] Anthropic. 2024. Claude 2.1: Enhanced capabilities and context length improvements. Anthropic News.

[2] Daehyeon Baek, Soojin Hwang, and Jaehyuk Huh. 2024. pSyncPIM: Partially Synchronous Execution of Sparse Matrix Operations for All-Bank PIM Architectures. In *Proceedings of the 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 354–367.

[3] Payman Behnam, Yaosheng Fu, Ritchie Zhao, Po-An Tsai, Zhiding Yu, and Alexey Tumanov. 2025. RocketKV: Accelerating Long-Context LLM Inference via Two-Stage KV Cache Compression. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*.

[4] Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. 2024. ArkVale: Efficient Generative LLM Inference with Recallable Key-Value Eviction. In *38th Conference on Neural Information Processing Systems (NeurIPS 2024)*.

[5] Google. 2018. CloudTPU. https://cloud.google.com/tpu/docs/system-architecture-tpu-vm.

[6] Yufeng Gu, Alireza Khadem, Sumanth Umesh, Ning Liang, Xavier Servot, Onur Mutlu, Ravi Iyer, and Reetuparna Das. 2025. PIM Is All You Need: A CXL-Enabled GPU-Free System for Large Language Model Inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25)*.

[7] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. 2024. NeuPIMs: A NPU-PIM Heterogeneous Acceleration for Batched Inference of Large Language Model. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*.

[8] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv preprint arXiv:2310.06825.

[9] Hongju Kal, Chanyoung Yoo, and WonWoo Ro. 2023. AESPA: Asynchronous Execution Scheme to Exploit Bank-Level Parallelism of Processing-in-Memory. In *56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[10] Hyucksung Kwon, Kyungmo Koo, Janghyeon Kim, Woongkyu Lee, Minjae Lee, Gyeonggeun Jung, Hyungdeok Lee, Yousub Jung, Jaehan Park, Yosub Song, Byeongsu Yang, Haerang Choi, Guhyun Kim, Jongsoon Won, Woojae Shin, Changhyun Kim, Gyeongcheol Shin, Yongkee Kwon, Ilkon Kim, Euicheol Lim, John Kim, and Jungwook Choi. 2026. PIMphony: Overcoming Bandwidth and Capacity Inefficiency in PIM-based Long-Context LLM Inference System. In *2026 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[11] 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, Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Myeongjun Lee, Minyoung Shin, Minhwan Shin, Jaekyung Cha, Changson Jung, Kijoon Chang, Chunseok Jeong, Euicheol Lim, Il Park, Junhyun Chun, and Sk Hynix. 2022. System Architecture and Software Stack for GDDR6-AiM. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. 1–25.

[12] Hyojung Lee, Daehyeon Baek, Jimyoung Son, Jieun Choi, Kihyo Moon, and Minsung Jang. 2025. PAISE: PIM-Accelerated Inference Scheduling Engine for Transformer-based LLM. In *2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[13] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 43–56.

[14] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[15] Cong Li, Yihan Yin, Xintong Wu, Jingchen Zhu, Zhutianya Gao, Dimin Niu, Qiang Wu, Xin Si, Yuan Xie, Chen Zhang, and Guangyu Sun. 2025. H^2-LLM: Hardware-Dataflow Co-Exploration for Heterogeneous Hybrid-Bonding-based Low-Batch LLM Inference. In *Proceedings of the 2025 ACM/IEEE 52nd Annual International Symposium on Computer Architecture (ISCA)*.

[16] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob. 2020. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. 19, 2 (2020), 106–109.

[17] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. SnapKV: LLM Knows What You are Looking for Before Generation. In *38th Conference on Neural Information Processing Systems (NeurIPS 2024)*.

[18] Chaoqiang Liu, Haifeng Liu, Dan Chen, Yu Huang, Yi Zhang, Wenjing Xiao, Xiaofei Liao, and Hai Jin. 2025. HeterRAG: Heterogeneous Processing-in Memory Acceleration for Retrieval-augmented Generation. In *Proceedings of the 2025 ACM/IEEE 52nd Annual International Symposium on Computer Architecture (ISCA)*.

[19] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. Scissorhands: Exploiting the Persistence of Importance Hypothesis for LLM KV Cache Compression at Test Time. In *37th Conference on Neural Information Processing Systems (NeurIPS 2023)*.

[20] Stefan Mach, Fabian Schuiki, Florian Zaruba, and Luca Benini. 2021. FPnew: An Open-Source Multiformat Floating-Point Unit Architecture for Energy-Proportional Transprecision Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29, 4 (2021), 774–787.

[21] Meta. 2024. Introducing Llama 3.1. Meta AI Blog.

[22] Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7.0: A Tool to Model Caches/Memories, 3D stacking, and off-chip IO. https://github.com/HewlettPackard/cacti.

[23] OpenAI. 2023. Introducing ChatGPT Enterprise. OpenAI Blog.

[24] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn. 2024. AttAcc! Unleashing the Power of PIM for Batched Transformer-based Generative Model Inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*.

[25] B. S. Premananda, M. K. Chandana, K. P. Shree Lakshmi, and A. M. Keerthi. 2017. Design of low power 8-bit carry select adder using adiabatic logic. In *2017 International Conference on Communication and Signal Processing (ICCSP)*.

[26] Derrick Quinn, E. Ezgi Y ucel, Kim Jinkwon, José F. Martínez, and Mohammad Alian. 2025. LongSight: Compute-Enabled Memory to Accelerate Large-Context LLMs via Sparse Attention. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[27] Minseok Seo, Xuan Truong Nguyen, Seok Joong Hwang, Yongkee Kwon, Guhyun Kim, Chanwook Park, Ilkon Kim, Jaehan Park, Jeongbin Kim, Woojae Shin, Jongsoon Won, Haerang Choi, Kyuyoung Kim, Daehan Kwon, Chunseok Jeong, Sangheon Lee, Yongseok Choi, Wooseok Byun, Seungcheol Baek, Hyuk-Jae Lee, and John Kim. 2024. IANUS: Integrated Accelerator based on NPU-PIM Unified Memory System. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*.

[28] Hyojun Son, Gilbert Jonatan, Wu Xiangyu, Haeyoon Cho, Kaustubh Shivdikar, José L. Abellán, Ajay Joshi, David Kaeli, and John Kim. 2025. PIMnet: A Domain-Specific Network for Efficient Collective Communication in Scalable PIM. In *2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[29] Chirag Sudarshan, Mohammad Hassani Sadi, Christian Weis, and Norbert Wehn. 2022. Optimization of DRAM based PIM Architecture for Energy-Efficient Deep Neural Network Training. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*.

[30] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 2024. Quest: Query-Aware Sparsity for Efficient Long-Context LLM Inference. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*.

[31] Gemini Team. 2024. Gemini 1.5: Unlocking Multimodal Understanding Across Millions of Tokens of Context. arXiv preprint arXiv:2403.05530.

[32] 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. [n. d.]. Llama 2: Open Foundation and Fine-Tuned Chat Models.

[33] Haoran Wang, Yuming Li, Haobo Xu, Ying Wang, Liqi Liu, Jun Yang, and Yinhe Han. 2025. LAD: Efficient Accelerator for Generative Inference of LLM with Locality Aware Decoding. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[34] Zhiyu Wu, Xiaokang Chen, Zizheng Pan, Xingchao Liu, Wne Liu, Damai Dai, Huazuo Gao, Yiyang Ma, Chengyue Wu, Bingxuan Wang, Zhenda Xie, Ye Wu, Kai Hu, Jiawei Wang, Yaofeng Sun, Yukun Li, Yishi Pia, Kang Guan, Aixin Liu, Xin Xie,

Yuxiang You, Kai Dong, Xingkai Yu, Haowei Zhang, Liang Zhao, Yisong Wang, and Chong Ruan. 2024. DeepSeek-VL2: Mixture-of-Experts Vision-Language Models for Advanced Multimodal Understanding. arXiv preprint arXiv:2412.10302.

[35] Xinfeng Xie, Peng Gu, Yufei Ding, Dimin Niu, Hongzhong Zheng, and Yuan Xie. 2023. MPU: Memory-centric SIMT Processor via In-DRAM Near-bank Computing. *ACM Transactions on Architecture and Code Optimizations* 20, 3 (2023).

[36] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 570–583. https://doi.org/10.1109/HPCA51647.2021.00055

[37] Qingyue Yang, Jie Wang, Xing Li, Zhihai Wang, Chen Chen, Lei Chen, Xianzhi Yu, Wulong Liu, Jianye Hao, Mingxuan Yuan, and Bin Li. 2025. AttentionPredictor: Temporal Patterns Matter for KV Cache Compression. In *39th Conference on Neural Information Processing Systems (NeurIPS 2025)*.

[38] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Rè, Clark Barret, Zhangyang Wang, and Beidi Chen. 2023. H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *37th Conference on Neural Information Processing Systems (NeurIPS 2023)*.