PIMPAL: Accelerating LLM Inference on Edge Devices via In-DRAM Arithmetic Lookup

Yoonho Jang*, Hyeongjun Cho†, Yesin Ryu‡§, Jungrae Kim*, Seokin Hong*
*Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon, Republic of Korea
†Department of Semiconductor Convergence Engineering, Sungkyunkwan University, Suwon, Republic of Korea
‡Department of Semiconductor and Display Engineering, Sungkyunkwan University, Suwon, Republic of Korea
§Samsung Electronics, Hwaseong, Republic of Korea
{jyh8807, cho0624ck}@skku.edu {seleneyou}@g.skku.edu {dale40, seokin}@skku.edu

***Abstract*—Deploying Large Language Models (LLMs) on edge devices poses significant challenges due to their high computational and memory demands. In particular, General Matrix-Vector Multiplication (GEMV), a key operation in LLM inference, is highly memory-intensive, making it difficult to accelerate using conventional edge computing systems. While Processing-in-memory (PIM) architectures have emerged as a promising solution to this challenge, they often suffer from high area overhead or restricted computational precision.**

This paper proposes *PIMPAL (Processing-In-Memory architecture with Parallel Arithmetic Lookup)*, a cost-effective PIM architecture leveraging LookUp Table (LUT)-based computation for GEMV acceleration in sLLMs (small LLMs). By replacing traditional arithmetic operations with parallel in-DRAM LUT lookups, PIMPAL significantly reduces area overhead while maintaining high performance. PIMPAL introduces three key innovations: (1) it divides DRAM bank subarrays into compute blocks for parallel LUT processing; (2) it employs *Locality-aware Compute Mapping (LCM)* to reduce row activations by maximizing LUT access locality; and (3) it enables multi-precision computations through a *LUT Aggregation (LAG)* mechanism that combines results from multiple small LUTs. Experimental results show that PIMPAL achieves up to 17.8x higher performance than previous LUT-based PIM designs and reduces area overhead by 40% compared to conventional processing unit-based PIM designs.

I. Introduction

Transformer-based Large Language Models (LLMs) are driving significant advancements in Natural Language Processing (NLP) and transforming diverse applications. However, their massive computational and memory requirements restrict deployment to high-performance data centers equipped with specialized hardware like GPUs and TPUs. To address these challenges, Small LLMs (sLLMs) have emerged as compact alternatives [22, 25, 26, 29]. These lightweight models significantly reduce parameter counts compared to full-scale LLMs, making LLM inference feasible on resource-constrained devices such as smartphones and IoT systems.

Although sLLMs are compact and efficient, they still face significant challenges in accelerating General Matrix-Vector Multiplication (GEMV), a core operation in both LLMs and sLLMs. GEMV is memory-intensive, making it difficult to optimize on conventional computing systems with limited memory bandwidth [4, 27, 28]. In data centers, LLMs mitigate this bottleneck by replacing GEMV with General Matrix-Matrix Multiplication (GEMM), which is a compute-intensive alternative, through input batching. However, this optimization is infeasible on edge devices, where only a single input is typically available during inference.

Recently, many prior works have proposed Processing-In-Memory (PIM) architectures to accelerate memory-intensive workloads, such as GEMV, by performing computations within memory to exploit high internal bandwidth [1, 5, 9, 15, 20, 21]. One notable example is the Processing Unit (PU)-based approach [9, 15], where numerous PUs are integrated within or near memory banks. Prototype devices developed on commodity DRAMs like HBM2 [15] and GDDR6 [9] demonstrate the significant acceleration of GEMV for deep neural network workloads. However, despite these performance gains, PU-based PIM incurs substantial area overhead, reducing memory capacity by 50% in the prototypes. This overhead mainly arises from the inherent differences between memory and logic fabrication processes, such as larger feature sizes and fewer metal layers in memory-focused processes [5].

Another PIM architecture, LookUp Table (LUT)-based PIM [6, 23, 30], performs computations by leveraging precomputed results stored in a LUT within memory. Unlike PU-based approaches, LUT-based PIM does not require additional processing units; instead, it directly looks up values stored in memory cells, enabling the implementation of complex functions with minimal area overhead. Furthermore, by modifying the values stored in the LUT, this architecture can flexibly support a wide range of functions without requiring changes to the hardware design [6, 30]. These advantages make LUT-based PIM a cost-effective and versatile solution.

However, LUT-based PIM faces two major challenges. First, storing LUTs in DRAM cells often leads to frequent row activations, which are time-consuming and power-intensive [8, 17], especially when large LUTs span multiple DRAM rows. This problem is exacerbated when the inputs to the arithmetic function are randomly distributed, resulting in poor row buffer locality. Second, supporting high-precision arithmetic requires large LUTs. For instance, a LUT for multiplying two 8-bit integers requires 128KB ($2^8 \times 2^8 \times 16$ bits), which far exceeds the typical DRAM row size (i.e., 1 $\sim$2KB). As a result, such large LUTs must be distributed across multiple rows, further increasing row activation frequency.

This paper presents ***PIMPAL***, a novel LUT-based PIM

architecture specifically designed to accelerate the GEMV operations of sLLMs. PIMPAL employs a minimal number of PUs and replaces complex operations with highly parallel LUT lookups. By replacing substantial portions of the PUs with LUTs, it achieves high performance with low area overhead. PIMPAL addresses the key challenges of prior LUT-based PIM architectures via its parallel organization and three novel mechanisms as follows:

Organization for Subarray-level Parallel Lookup: PIMPAL divides a DRAM bank's subarray into compute blocks, each consisting of multiple subarrays. Within each compute block, one subarray is responsible for supplying data, while another subarray is dedicated to storing the LUT. This organization simplifies LUT access and ensures efficient data retrieval. By enabling parallel processing across multiple compute blocks, PIMPAL provides a scalable framework that supports high-throughput computation.

Locality-aware Compute Mapping: To minimize DRAM row activations during GEMV computations, PIMPAL employs *Locality-aware Compute Mapping (LCM)*. This mechanism uses a column-major GEMV approach, where each vector element is multiplied by all elements in its corresponding matrix column. LCM distributes matrix columns across compute blocks, assigning each block to calculate individual partial products. This mapping allows an activated LUT row for a vector element to be reused multiple times, significantly improving the locality of LUT data stored in the DRAM row buffer within each compute block.

LUT Aggregation: To support multi-precision operations such as INT8 and BF16, PIMPAL introduces the *LUT Aggregation (LAG)* mechanism. For floating-point operations, LAG combines lookup results from two LUTs for exponential and mantissa bits. By using multiple small LUTs, PIMPAL supports high-precision computations while significantly reducing LUT size. For example, the BF16 LUT entries are reduced from 2^{32} elements to 2^{16} elements. When the LUTs are small enough to fit within a single subarray, PIMPAL stores them together and enables partial row activation to access individual LUTs efficiently, minimizing area overhead.

Our experimental results show that PIMPAL achieves a 17x speedup over previous LUT-based PIM architectures and delivers nearly 25% higher performance per area compared to PU-based PIM architectures for GEMV operations in sLLMs.

II. Background and Motivation

A. GEMV Operation in LLMs

LLMs are highly memory-intensive due to their architectural design, which relies heavily on fully connected (FC) layers implemented using General matrix-vector multiplication (GEMV). Although GEMV involves simple Multiply-and-Accumulate (MAC) computations, its low data reuse rate imposes substantial memory bandwidth demands. Table I shows the proportion of inference time spent on FC layers for various sLLMs on the NVIDIA A100 GPU. Despite the high-performance GPU's use of high-bandwidth memory, FC layers account for over 60% of the total model inference time. These experimental results indicate that GEMV constitutes the majority of sLLM inference workloads, emphasizing the need to optimize this operation for better performance.

TABLE I: The sizes of sLLMs and the proportion of GEMV in their inference time when using one input on an Nvidia A100 GPU.

	OPT-350m	**Gemma-2**	**Llama-3.2**	**MobileBert**
Model size	700 MB	2 GB	2 GB	101 MB
FC layers	61.2 %	62.4 %	63.5 %	73.2 %

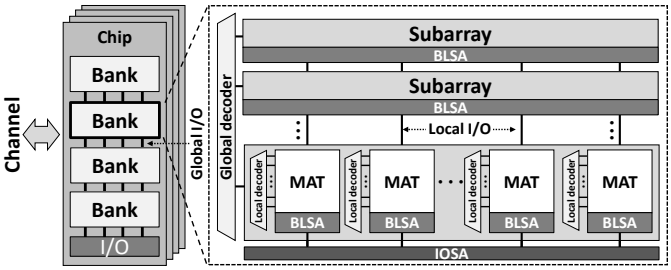


Fig. 1: DRAM Organization.

B. DRAM Organization and Operations

Figure 1 illustrates the hierarchical structure of DRAM. A DRAM channel is a logical construct defined by the memory controller that interfaces with memory. Each channel operates independently, allowing for parallel data transfer. A channel can be connected to one or more DRAM chips, depending on the memory module configuration and standards like DDR, GDDR, or LPDDR. Each chip consists of multiple banks, which can be accessed independently to increase memory bandwidth through bank-level parallelism. Each bank's memory array is further divided into subarrays.

When accessing data, a global decoder selects a subarray, and a local decoder activates a specific row within it. Each subarray consists of multiple MATs, which provide data from the activated row. The row's data is latched in the Bitline Sense Amplifiers (BLSA or Row Buffer). To read or write data, a column decoder selects the column in the BLSA, and the MATs transfer data to or from the IOSA via the Local I/O path. When a row within a bank is activated, it must be precharged to activate another row. However, if the requested row is already activated and its data resides in the BLSA, subsequent accesses bypass the activation and precharge steps. This optimized access, known as a row buffer hit, significantly reduces latency and is essential for achieving low data access times and reducing energy consumption.

C. Limitations of Prior PIM Architectures

1) PU-based PIM Architectures: Processing Unit (PU)-based PIM architectures integrate numerous processing units within or near memory banks, enabling data to be accessed directly from memory cells without relying on the external I/O bus. This design leverages the massive internal memory bandwidth of DRAM, achieving state-of-the-art computation speeds for GEMV acceleration. Recently, DRAM vendors have prototyped PU-based PIM devices using commodity DRAMs, such as HBM2 [15] and GDDR6 [9], demonstrating their potential for high-performance computing applications.

Permanent Capacity Loss: Despite their high performance, these architectures sacrifice approximately 50% of memory

TABLE II: Comparison of LUT-based PIM architectures.

Architecture	LUT Storage	# of row activation per 2K MUL.	LUT bit Precision	INT8 / BF16 GEMV Support
pPIM [23]	Additional Buffer	None	8-bit	No
pLUTo [6]	DRAM cells	256	8-bit	No
ReD-LUT [30]	DRAM cells	1 to 2K	8-bit	No
PIMPAL (This work)	DRAM cells	1	16-bit	Yes

capacity to accommodate processing units [13, 14, 15]. This significant area overhead arises because logic circuits require larger space when implemented using a DRAM fabrication process compared to the standard logic fabrication process [5]. The integration of these processing units occupies space otherwise allocated to memory cells, resulting in a permanent reduction in capacity. To compensate for this loss, additional DRAM chips are required to meet capacity demands, which poses challenges for systems with constrained physical space for memory modules, such as edge devices.

2) LUT-based PIM Architecture: LUT-based PIM architectures leverage Look-Up Tables (LUTs) to enable complex computations with minimal changes to DRAM organization. These architectures store pre-computed data for specific functions and retrieve results directly from the tables, requiring only storage for LUT elements and simple decoding logic. This flexible approach supports various functions by writing different data into the tables. Given these advantages, several LUT-based PIM architectures have been proposed, as summarized in Table II. pPIM adds buffers for LUT storage near DRAM subarray and retrieves elements using operands from BLSA [23]. In contrast, pLUTo [6] and ReD-LUT [30] store LUTs in DRAM cells but access them differently. pLUTo activates all LUT rows in a subarray and reads multiple LUT elements by matching operand values with row numbers, while ReD-LUT activates a single row using one operand as the row address and retrieves the specific element by using another operand as the column address.

Frequent Row Activation: Prior LUT-based PIM designs suffer from low performance due to frequent DRAM row activations, which are time-consuming operations in DRAM systems. A key issue is that these designs fail to exploit row buffer hits during LUT access. For example, pLUTo's row-sweeping operation activates all LUT rows, causing excessive DRAM row activations. Similarly, ReD-LUT activates a single row per computation but produces only one element per activation, leading to high row activation overhead for large datasets. As discussed in Section II-B, DRAM systems can bypass row activation if the requested row is already activated. Therefore, improving spatial and temporal localities of LUT elements within the row buffer is essential to reducing DRAM row activations and enhancing performance.

Low Bit-Precision: Another critical challenge is the limited bit-precision of LUTs, which restricts their applicability to typical DNN workloads. Most models require at least 8-bit MAC operations, as INT8 is the standard arithmetic precision for inference tasks, necessitating a 16-bit LUT to handle the multiplication of two 8-bit operands. However, prior PIM architectures provide only 8-bit LUTs, which is insufficient for core NN operations. While pPIM achieves 8-bit MAC operations by combining outputs, it introduces significant area overhead due to the need for numerous latches and complex routing logic. The increasing adoption of BF16 precision, which delivers FP32-level accuracy in 16 bits, further exacerbates this issue, as it demands large LUTs with massive entry sizes ($2^{16} \times 2^{16} \times 16$-bit = 8GB). Hence, it is essential to provide high bit-precision LUT with low area overhead.

III. PIMPAL

This section introduces ***PIMPAL*** *(Processing-In-Memory architecture with Parallel Arithmetic Lookup)*, a novel LUT-based PIM architecture specifically designed to accelerate the GEMV operations. PIMPAL is structured to have a parallel organization for enabling parallel lookup within a DRAM bank and incorporates two novel mechanisms: *Locality-aware Compute Mapping (LCM)* and *LUT Aggregation (LAG)*.

A. Locality-aware Compute Mapping

To minimize DRAM row activations during GEMV computations, PIMPAL employs *Locality-aware Compute Mapping (LCM)*, which utilizes a column-major GEMV approach illustrated in Figure 2. In this approach, each matrix column is multiplied by its corresponding vector element (steps ❶ to ❸) to generate partial products, which are then accumulated to produce the final results. Using this method, LCM activates a DRAM row containing LUT row for a given vector element (e.g., value 2), as shown in Figure 3. It then retrieves LUT elements from the activated row consecutively for all matrix elements (e.g., values 3, 1, and 2) of the corresponding column. This approach limits LUT row activations to the vector size, as a vector element is reused across all elements in its matrix column during each step. For the example in Figure 2, LCM involves only three LUT row activations.

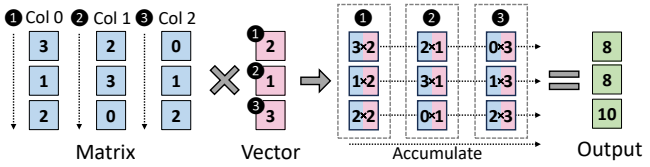


Fig. 2: Column-major GEMV.

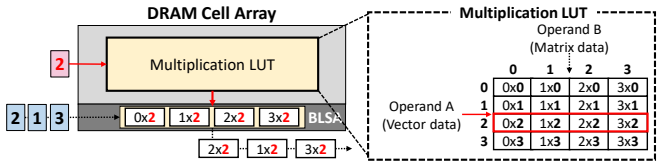


Fig. 3: Locality-aware Compute Mapping leveraging Column-major GEMV. The DRAM row size is assumed to be the same as the LUT row size.

B. PIMPAL Organization and Operation

PIMPAL organizes each DRAM bank into compute blocks, where two subarrays are grouped to form one compute block. Within a compute block, one subarray supplies input data for computations, while the other provides Look-Up Table (LUT) elements, enabling efficient data retrieval and processing. This organization also maximizes computation throughput with parallel LUT lookup by exploiting subarray-level parallelism [11].

Figure 4 illustrates PIMPAL's bank architecture. Each compute block comprises two adjacent subarrays, one storing

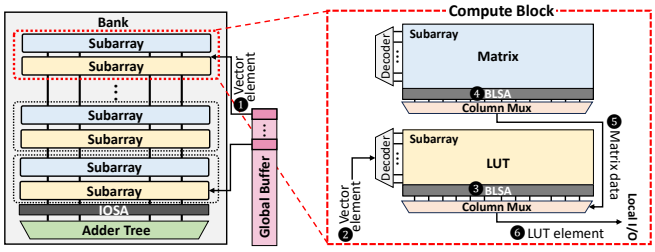


Fig. 4: Bank architecture of PIMPAL.

matrix data and the other storing LUT elements. A global buffer, shared across banks, contains vector elements that are distributed to the compute blocks within the banks (❶). For each vector element (❷), a DRAM row in the LUT subarray is activated, latching the corresponding LUT row in the BLSA (❸). Next, a row in the matrix subarray, storing the matrix column to be multiplied by the vector element, is activated (❹). The selected matrix column element is then used to retrieve a specific LUT element from the BLSA of the LUT subarray (❺). The retrieved LUT element is transferred via the local I/O path to the IOSA (❻). Since the LUT element size (e.g., 16-bit) is smaller than the typical column size, each compute block uses only a subset of the available local I/O path lines. The IOSA receives LUT elements from the compute blocks and transfers them into the adder tree for fast accumulation.

Figure 5 shows overall GEMV processing in compute block 0 for the example GEMV shown in Figure 2. As illustrated in the Figure, the compute block 0 is assigned the computation for matrix column 0. Thus, the first element of the vector is used as the LUT row address. As the value of the element is 2, the DRAM row containing the LUT row 2 is activated (①). Subsequently, the compute block activates the DRAM row corresponding to the matrix column correlated with the vector element in the matrix subarray (②). After activating these two rows, the compute block decodes the matrix column data and transfers it to the LUT subarray via an additional local path. Finally, the column decoder of the LUT subarray uses the matrix column data to select the corresponding element (③) and sends it to the adder tree for accumulation alongside the partial products computed by other compute blocks.

The accumulator in PIMPAL utilizes a specialized adder tree inspired by Newton [13]. This adder tree aligns the mantissa of the multiplier results with the largest exponent among them, allowing the mantissa to be accumulated as integers. This design facilitates fast processing while maintaining a minimal accuracy drop. Furthermore, the adder tree employs 24-bit adders, enabling compatibility with BF16 and 16-bit integer addition when INT8 precision is used as the mantissa.

C. LUT Aggregation

To support high-precision computations while minimizing LUT size and area overhead, PIMPAL employs the *LUT Aggregation (LAG)* mechanism. LAG decomposes complex arithmetic operations, such as high-precision multiplication, into smaller sub-operations processed by multiple small LUTs, then aggregates lookup results from the tables to achieve the desired precision. With this approach, PIMPAL enables efficient support for high-precision arithmetic like BF16. This subsection explains the arithmetic decomposition and the mapping process of multiple LUTs to compute blocks.

1) Arithmetic Decomposition: Floating-point operations, such as multiplication, involve independent calculations for the mantissa and exponent components of the operands. This property enables an efficient Arithmetic Decomposition technique, where separate LUTs are used for each component. Specifically, a dedicated LUT handles the mantissa multiplication, while another handles the exponent addition and bias adjustment. By splitting the computation in this manner, the complexity and size of each LUT are significantly reduced. For example, a BF16 LUT is decomposed into two separate LUTs (Mantissa LUT and Exponent LUT), as shown in Figure 6. With this decomposition, the total size of BF16 LUT is reduced from 8GB to 128KB.

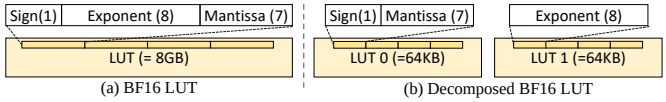


Fig. 6: LUT size reduction with arithmetic decomposition. Arithmetic decomposition reduces the LUT size for BF16 multiplication from 8GB to 128KB.

2) LUT Mapping: We now describe how the LUTs are mapped to compute blocks for different precisions. As described above, the arithmetic decomposition splits a LUT to the exponent and mantissa LUTs. Figure 7a illustrates the data format of LUT elements for INT16 and BF16 data. Figure 7b and Figure 7c show how the compute block stores the LUTs and accesses them for INT16 and BF16 data, respectively. For this explanation, we assume a subarray configuration consisting of 1024 rows, each with a 2KB capacity.

INT16: For supporting the precise multiplication of INT8 operands, it is necessary to support 16-bit precision. Therefore,

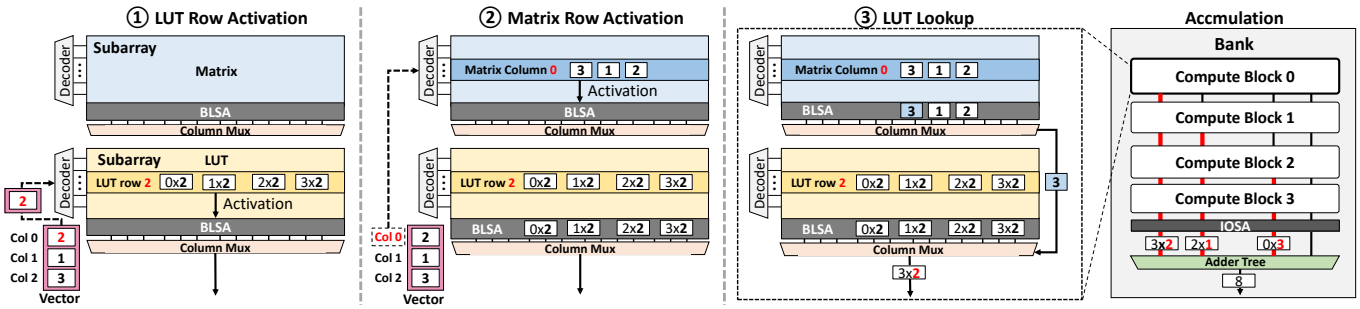


Fig. 5: Overall operation of GEMV processing in the compute block and partial product accumulation. The operation sequence proceeds from left to right. We assume three compute blocks are involved in the processing.

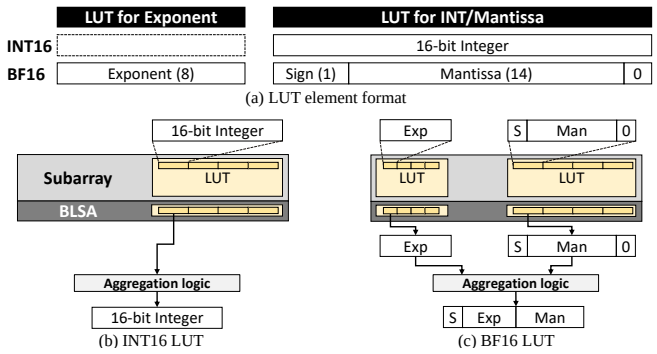


Fig. 7: Multi-precsion support with different LUT mappings within compute block.

PIMPAL stores the LUT elements as 16-bit values. These elements function as the mantissa in the adder tree. Since integer types have no exponent value, they do not use a LUT for the exponent part. This 16-bit LUT can also be used for any function requiring 16-bit data for a single operand. As the size of 16-bit LUT is 128KB ($2^8 \times 2^8 \times$16-bit), the compute block needs only one subarray for storing the LUT (Figure 7b).

BF16: To store BF16 data in the LUT, we must accommodate its 8-bit exponent and 7-bit mantissa. The exponent LUT utilizes an 8-bit format, as the exponent cannot exceed 8 bits in floating point operations. Meanwhile, we employ a 14-bit format for the mantissa LUT to cover the 7-bit mantissa multiplication and account for potential rounding during computations. Additionally, we add a 1-bit sign in the mantissa, resulting in a 15-bit format for the mantissa LUT entries. Adding the sign bit does not significantly increase the number of mantissa LUT entries, as it only requires $2^8 \times 2^8$ entries. To facilitate easy decoding of multiple data formats, we zero-pad the mantissa elements to make them a multiple of 8 bits, resulting in a 16-bit type. Like the INT16 case, a single subarray can store both the mantissa LUT and the exponent LUT (Figure 7c). The compute block looks up these two LUTs and then aggregates the results retrieved from the LUTs to construct the final BF16 result.

As observed above, the compute block stores the LUT only part of the subarray in most cases. To activate the needed part of a DRAM row to fetch a LUT row, we employ a technique called Partial Row Activation (PRA) [16], which allows activation of part of the row within the subarray. PRA activates the row segments in the MATs where the corresponding bit in the PRA latch is set to 1. We refer to this method as Partial LUT Activation (PLA), as depicted in Figure 8. PLA enables multiple LUT rows in different parts to be activated sequentially, preventing LUT rows from overwriting each other.

D. Optimization for LUT fetching

Before starting GEMV operations, all compute blocks require the LUT for multiplication. Storing identical LUT rows in each compute block is inefficient, reducing subarray capacity by 12.5% to 25% with a 16-bit LUT following the DRAM configuration. This overhead is especially problematic for normal memory operations, as it limits available capacity.

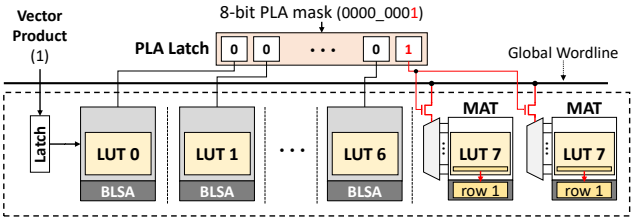


Fig. 8: Partial LUT Activation for fetching 8-bit LUT data. A dotted box represents a subarray.

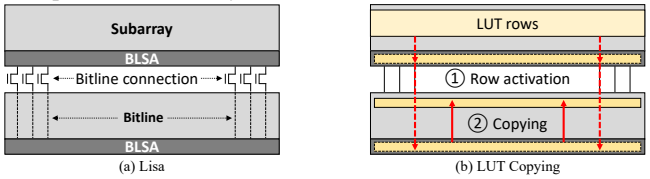


Fig. 9: (a) LISA technique [3] and (b) fast LUT copying with LISA. A red line represents a row activation.

To address this issue, we store the LUT in a single subarray per bank and use the Low-Cost Interlinked Subarrays (LISA) technique to copy it at the start of PIM operations. This technique interconnects the bitline of each subarray with another using a single transistor and transfers the activated row data into the connected subarray's BLSA, illustrated in Figure 9. By adopting this approach, the LUT copying step can be executed quickly at the start of the PIM operation.

IV. Evaluation

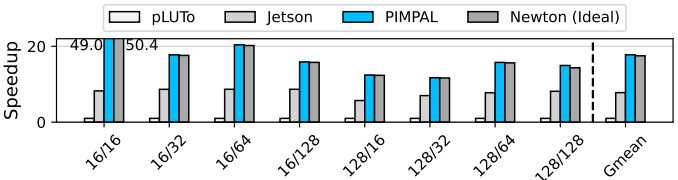
A. Methodology

To evaluate the performance of PIMPAL, we implement PIM architectures on Ramulator, a cycle-level DRAM simulator [7, 12], following the configuration of a memory product used in edge devices (Table III) [10]. We compare two PIM architectures with PIMPAL: Newton and pLUTo. We configure them with the same memory configuration for fairness. We also compare PIMPAL with NVIDIA Jetson Xavier GPU [19], which is popularly used to accelerate NN tasks on edge devices. GPU performance metrics are gathered from real-time tests, with the configurations in Table III.

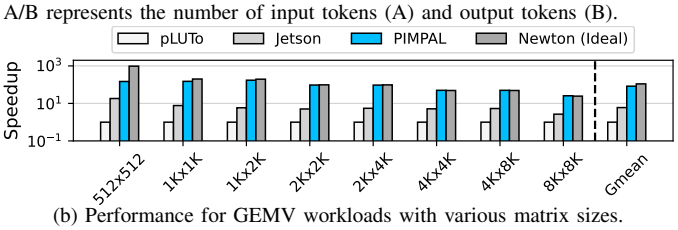
For workloads, we use Gemma-2 2B, a well-known sLLM from Google that supports multiple precisions (from FP32 to INT4) [25]. We evaluate the workload's performance by measuring the end-to-end inference time. For this evaluation, GEMV layers are processed with PIM, while other layers are processed using the Jetson platform. We also use synthetic GEMV workloads commonly found in LLMs. To ensure consistency, we use INT8 precision for Gemma-2, while pLUTo is evaluated at INT4 precision due to its limited LUT size.

TABLE III: Configurations.

LPDDR5-6400 Configuration	
Channel & Rank	4 & 1 per channel
Bank group & Bank	4 per channel & 4 per bank group
Subarray & Row size	64 per bank & 2KB
tCL-tRCD-tRP & tRRD, tFAW	17-18-18 & 5ns, 20ns
Technique	**Parallelism**
Newton (PU-based)	16 MAC units per bank
pLUTo (LUT-based)	32 LUT units per bank
PIMPAL (LUT-based)	16 Compute blocks per bank
NVIDIA Jetson AGX Xavier	
Core	512 CUDA & 64 Tensor core
Memory	64 GB



(a) End-to-end performance for various numbers of tokens. On the X-axis, A/B represents the number of input tokens (A) and output tokens (B).



(b) Performance for GEMV workloads with various matrix sizes.

Fig. 10: Performance Comparison.

B. Performance

Figure 10a shows the normalized performance relative to pLUTo for end-to-end inference tasks. The results indicate that pLUTo has the lowest performance due to frequent LUT row activations. In contrast, Newton and PIMPAL achieve substantial speedups of 17.5x and 17.8x over pLUTo, respectively. For GEMV workloads across various matrix sizes, both architectures show significant performance improvement, as shown in Figure 10b. Newton and PIMPAL have similar overall execution times, as both spend considerable time on operand and LUT element decoding, incurring tCCD DRAM cycles. In smaller GEMV workloads, Newton outperforms PIMPAL because PIMPAL cannot fully utilize open LUT rows with fewer operands. Fortunately, this performance impact is minimal for end-to-end inference due to the extensive number of FC layers with substantial size in LLMs, resulting in only a 2% performance difference between Newton and PIMPAL. Additionally, both architectures outperform the Jetson platform, achieving a 2.24x speedup for the end-to-end inference tasks. As shown in Figure 10b, both architectures efficiently accelerate GEMV, achieving average speedups of 18.3x and 13.8x over Jetson for GEMV workloads.

C. Area Overhead

Table IV presents the area breakdown of the PIMPAL and Newton architectures, both implemented using the same configuration as in the performance evaluation. PIMPAL includes additional components such as the LUT decoder, adder tree, and minor elements. Storage components are modeled using CACTI 7.0, and processing logic is synthesized with Synopsys Design Compiler [2, 18, 24]. By reducing the number of multipliers in the PU components by 40%, PIMPAL significantly lowers the PU area overhead. Figure 11 compares the performance per area of GEMV workloads relative to Newton. PIMPAL demonstrates higher performance per area overall,

TABLE IV: Area breakdown

Area (mm^2)	PIMPAL	Newton
Adder Tree	1.7936	1.7936
LUT decoder	0.0233	-
Multiplier	-	1.2464
Others	0.005215	0.003215
Total	**1.8222**	**3.0432**

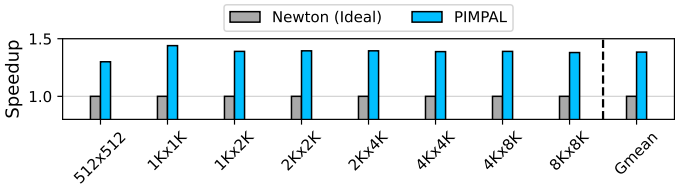


Fig. 11: Performance per area for GEMV workloads with various matrix sizes.

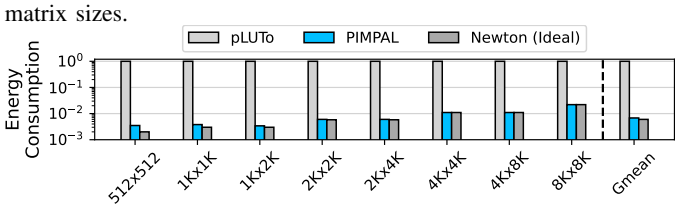


Fig. 12: Energy consumption comparison for GEMV workloads with various matrix sizes.

with a slight decrease in lower-dimension workloads due to the reasons discussed earlier. On average, PIMPAL achieves a 25% improvement in performance-per-area efficiency compared to the representative PU-based PIM architecture.

D. Energy Consumption

Given the importance of energy efficiency on edge devices, we evaluate the energy consumption of each PIM architecture normalized to pLUTo's results. As shown in Figure 12, pLUTo exhibits significantly higher energy consumption compared to the others due to its reliance on frequent row activations, whereas Newton and PIMPAL primarily involve a small number of row activations and column read operations. Although PIMPAL consumes slightly more energy than Newton because of LUT row activations, it mitigates this with the Partial LUT Activation technique, which minimizes row activation energy. As a result, PIMPAL achieves energy efficiency comparable to Newton, with only a 10% difference.

V. Conclusion

This paper presented PIMPAL, a LUT-based PIM architecture for accelerating GEMV operations in small Large Language Models (sLLMs). By introducing LUT Aggregation (LAG) to reduce LUT storage requirements and Locality-aware Compute Mapping (LCM) to minimize DRAM row activations, PIMPAL overcomes key limitations of prior LUT-based PIM designs. Experimental results demonstrate that PIMPAL achieves up to 17.8x higher performance than previous LUT-based designs and delivers 25% better performance-per-area efficiency compared to PU-based PIM architectures, making it a promising solution for high-precision computations in resource-constrained edge environments.

Acknowledgment

This work was partly supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP)-ITRC (Information Technology Research Center) grant funded by the Korea government (MSIT) (RS-2021-II212052, 25%), the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2023-00228970, 25%/RS-2022-II221170, 50%). Seokin Hong is the corresponding author.

REFERENCES

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 105–117. [Online]. Available: https://doi.org/10.1145/2749469.2750386

[2] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, jun 2017. [Online]. Available: https://doi.org/10.1145/3085572

[3] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 568–580.

[4] Z. Chishti and B. Akin, "Memory system characterization of deep learning workloads," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 497–505. [Online]. Available: https://doi.org/10.1145/3357526.3357569

[5] F. Devaux, "The true processing in memory accelerator," in *2019 IEEE Hot Chips 31 Symposium (HCS)*. Los Alamitos, CA, USA: IEEE Computer Society, aug 2019, pp. 1–24. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HOTCHIPS.2019.8875680

[6] J. D. Ferreira, G. Falcao, J. Gómez-Luna, M. Alser, L. Orosa, M. Sadrosadati, J. S. Kim, G. F. Oliveira, T. Shahroodi, A. Nori, and O. Mutlu, "pluto: Enabling massively parallel computation in dram via lookup tables," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 900–919.

[7] S. R. Group, "Ramulator: A dram simulator," https://github.com/CMU-SAFARI/ramulator, 2015.

[8] H. Ha, A. Pedram, S. Richardson, S. Kvatinsky, and M. Horowitz, "Improving energy efficiency of dram by exploiting half page row access," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[9] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. N. Vijaykumar, "Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 372–385.

[10] JEDEC, "Lpddr5," https://www.jedec.org/standards-documents/docs/jesd209-5c, 2023.

[11] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (salp) in dram," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. USA: IEEE Computer Society, 2012, p. 368–379.

[12] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[13] D. Kwon, S. Lee, K. Kim, S. Oh, J. Park, G.-M. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, V. Kornijcuk, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, B. An, J. Lee, D. Ko, Y. Jun, I. Kim, C. Song, I. Kim, C. Park, S. Kim, C. Jeong, E. Lim, D. Kim, J. Jang, I. Park, J. Chun, and J. Cho, "A 1ynm 1.25v 8gb 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops mac operation and various activation functions for deep learning application," *IEEE Journal of Solid-State Circuits*, vol. 58, no. 1, pp. 291–302, 2023.

[14] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, K. Vladimir, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, J. Lee, D. Ko, Y. Jun, K. Cho, I. Kim, C. Song, C. Jeong, D. Kwon, J. Jang, I. Park, J. Chun, and J. Cho, "A 1ynm 1.25v 8gb, 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops mac operation and various activation functions for deep-learning applications," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65, 2022, pp. 1–3.

[15] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware architecture and software stack for pim based on commercial dram technology : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 43–56.

[16] Y. Lee, H. Kim, S. Hong, and S. Kim, "Partial row activation for low-power dram system," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 217–228.

[17] Y. Lee, S. Kim, S. Hong, and J. Lee, "Skinflint dram system: Minimizing dram chip writes for low power," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 25–34.

[18] V. S. Naveen Muralimanohar, Ali Shafiee, "Cacti 7.0," https://github.com/HewlettPackard/cacti, 2017.

[19] Nvidia, "Nvidia jetson xavier," https://www.nvidia.com/ko-kr/autonomous-machines/embedded-systems/jetson-xavier-series/.

[20] S.-S. Park, K. Kim, J. So, J. Jung, J. Lee, K. Woo, N. Kim, Y. Lee, H. Kim, Y. Kwon, J. Kim, J. Lee, Y. Cho, Y. Tai, J. Cho, H. Song, J. H. Ahn, and N. S. Kim, "An lpddr-based cxl-pnm platform for tco-efficient inference of transformer-based large language models," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 970–982.

[21] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: in-memory accelerator for bulk bitwise operations using commodity dram technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 273–287. [Online]. Available: https://doi.org/10.1145/3123939.3124544

[22] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, and D. Zhou, "Mobilebert: a compact task-agnostic bert for resource-limited devices," 2020. [Online]. Available: https://arxiv.org/abs/2004.02984

[23] P. R. Sutradhar, S. Bavikadi, M. Connolly, S. Prajapati, M. A. Indovina, S. M. P. Dinakarrao, and A. Ganguly, "Look-up-table based processing-in-memory architecture with programmable precision-scaling for deep learning applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 263–275, 2022.

[24] Synopsys, "Synopsys design compiler," https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[25] G. Team, "Gemma 2: Improving open language models at a practical size," 2024. [Online]. Available: https://arxiv.org/abs/2408.00118

[26] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[27] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, apr 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

[28] W. Xu, Y. Zhang, and X. Tang, "Parallelizing dnn training on gpus: Challenges and opportunities," in *Companion Proceedings of the Web Conference 2021*, ser. WWW '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 174–178. [Online]. Available: https://doi.org/10.1145/3442442.3452055

[29] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," 2022.

[30] R. Zhou, A. Roohi, D. Misra, and S. Angizi, "Red-lut: Reconfigurable in-dram luts enabling massive parallel computation," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3508352.3549469