2026 59th IEEE/ACM International Symposium on Microarchitecture (MICRO)



PNMAX: Mapping-Architecture Co-Exploration Framework for Processing Near Memory

Chien-Yi Yang*
University of California, San Diego
La Jolla, USA
chy036@ucsd.edu

Jiantao Liu*
ETH Zürich
Zürich, Switzerland
jianliu@ethz.ch

Yue Pan
University of California, San Diego
La Jolla, USA
yup014@ucsd.edu

Tajana Rosing
University of California, San Diego
La Jolla, USA
tajana@ucsd.edu

Minxuan Zhou†
Illinois Institute of Technology
Chicago, USA
mzhou26@illinoistech.edu

Lana Josipović†
ETH Zürich
Zürich, Switzerland
ljosipovic@ethz.ch

***Abstract*—Processing near memory (PNM) mitigates the memory wall bottleneck in bandwidth-bound workloads, but its effectiveness depends on both the architectural design and the way a kernel is mapped onto hardware. However, existing PNM design exploration methods do not consider the complete mapping-architecture co-design space and lack a comprehensive PPA cost model, resulting in suboptimal designs. In this work, we introduce PNMAX, a novel design-space exploration (DSE) framework that effectively co-optimizes PNM mapping and architecture. PNMAX ensures the efficiency, effectiveness, and flexibility of mapping-architecture co-exploration through several key innovations, including: (1) a parameterized graph-based architecture specification capturing comprehensive hardware characteristics of PNM architectures; (2) a hardware-aware mapping formulation that concretely describes the hardware-dependent mapping choices; and (3) a bottom-up cost and constraint modeling framework that derives comprehensive and composable Performance, Power, and Area (PPA) costs. We exploit PNMAX to jointly optimize hardware parameters and software mappings of various existing and future PNM systems. Evaluations across representative AI/ML workloads demonstrate superior mapping quality and hardware efficiency of designs generated by PNMAX. We also exploit PNMAX to co-optimize mapping and architecture under unconventional architectural configurations, revealing several novel insights for pathfinding future PNM systems.**

***Index Terms*—processing near memory, PNM, design space exploration, DSE, software-hardware co-design**

I. Introduction

Processing in memory (PIM) is a promising solution to the memory bandwidth bottleneck imposed by deep neural networks (DNNs) and large language models (LLMs) [1]–[6], whose data movement between memory and compute dominates latency and energy in von Neumann architectures [7]–[10]. By migrating computation into or near memory arrays [11], PIM reduces data movement, exploits high internal memory bandwidth, and yields substantial energy-efficiency gains [6], [12]–[14]. Among PIM architectures, DRAM-based processing near memory (PNM) [9], [10], [15]–[37] is the most commercially mature [7], with industrial projects showing significant performance and energy gains over GPUs for data-intensive tensor workloads [9], [17], [38], [39]. Given its widespread use and relevance, this paper focuses on PNM.

Researchers have improved PNM systems from two perspectives *in isolation*: *architectural design* [9], [10], [17] (Figure 1) and *software kernel mapping* [7], [40] (e.g., how tensor kernels are scheduled, partitioned, and physically laid out in the PNM system). However, co-optimizing mapping and architecture is non-trivial: on the one hand, generating the optimal mapping relies on accurate costs and constraints derived from architectural characteristics, such as processing unit design, memory organization, data transfer links; on the other, architecture optimization must consider the optimal mappings based on compound metrics, such as performance, memory footprint, and power efficiency, for target workloads. Unfortunately, existing PNM design space exploration either focuses on mapping or architecture optimization, lacking the support for efficient and effective mapping-architecture co-optimization. Moreover, PNM designs are heavily constrained by power and area budgets [10]; optimizing for performance alone risks selecting architecturally unrealistic configurations where gains in performance come at the cost of energy consumption and memory footprint. However, no prior work accounts for the full tradeoff of Performance, Power, and Area (PPA) and memory footprint when co-optimizing the hardware and workload mapping on PNM architectures [7], [40], [41].

Table I compares the comprehensiveness of existing frameworks for PPA-aware mapping-architecture co-optimization on PIM architectures. CiMLoop [42] targets analog crossbar-based compute-in-memory, whose design space differs substantially from PNM. OptiPIM [40] and UniNDP [7] are closer in scope, but both fall short of enabling full mapping-architecture co-exploration under PPA and memory footprint tradeoffs. Specifically, OptiPIM uses a nested-loop mapping representation, constrained by fixed architectural assumptions,

This work was supported in part by PRISM and CoCoSys, centers in JUMP 2.0, an SRC program sponsored by DARPA, NSF grant #2112665, #2451744, and by Swiss National Science Foundation (grant number 215747).
* Equal contribution. † Co-corresponding authors.

2379-3155/26/$31.00 ©2026 IEEE
DOI 10.1109/MICRO67795.2026.00099

TABLE I

COMPARISON WITH PREVIOUS PIM DSE FRAMEWORK (◯ COVERS COMPREHENSIVE DESIGN SPACE, △ COVERS SUBSET OF DESIGN SPACE, × NO SUPPORT).

DSE Work	PNM Architecture	Explicit Mapping Space	PPA Exploration	Fast Cost Model	HW + SW Co-optimization
CiMLoop [42]	×	×	◯	◯	◯
UniNDP [7]	△	×	×	◯	×
OptiPIM [40]	△	×	×	△	×
PNMAX	◯	◯	◯	◯	◯

to generate high-performance mappings. Such a representation confines mapping decisions to the bank level, causing distinct data layout choices higher in the memory hierarchy to become indistinguishable in the representation. UniNDP decomposes tensor kernels into sequences of independent GEMM/GEMV sub-operations [43], which prevents data reuse across sub-operations from being expressed as an explicit mapping decision. Whether inputs shared across sub-operations are reused or duplicated in memory is not a tunable knob in the representation, leaving this choice outside the mapper's control. As a result, UniNDP's mapper operates on a reduced search space. Moreover, its tree-based architecture abstraction remains insufficiently parametric to represent the full PNM design space (Figure 2). These issues result in sub-optimal mapping-architecture decisions, as shown in Section IX.

An efficient and effective PNM mapping-architecture co-exploration framework must provide four key capabilities simultaneously: (1) an expressive, parameterized *architecture representation* that captures diverse hardware design dimensions, such as memory topology, PU placement, buffer sizing, and interconnect structure; (2) a comprehensive, architecture-aware *mapping representation* that jointly and explicitly models operation scheduling and physical data layout decisions across the full memory hierarchy; (3) a composable *PPA evaluation framework* that accurately reflects how different design choices in mapping and architecture impact the overall system performance and efficiency; and (4) an effective *mapping and architecture co-optimizer* that rapidly explores the design candidates across large architecture-mapping search spaces.

To address these challenges, we present PNMAX, a flexible and extensible co-exploration framework for PNM-based acceleration of **affine tensor kernels**, encompassing the above capabilities via several novelties. Specifically, PNMAX integrates a parameterized graph representation, GPAM, that captures the comprehensive PNM architecture design space. Unlike the tree-based representation of UniNDP [7], GPAM supports more flexible interconnect structures within memory while abstracting key PNM features as parameters for more concise cost modeling. Furthermore, in PNMAX, we propose a mapping representation that explicitly parameterizes architecture-dependent mapping decisions, thereby avoiding ambiguity and inaccuracy caused by prior nested-loop-based representations [40], [42]. To provide accurate PPA evaluation for any design point in the extensive mapping-architecture design space, we develop a *PPA extraction engine* that adopts an automatic, bottom-up approach to extract composable and reusable latency, power, and area characteristics across various parameterizable PNM components, including processing units, buffers, DRAM banks, interconnects, and control logic. Building upon the novel representations and PPA extraction engine, we develop a comprehensive cost and constraint modeling framework, which can be integrated with various optimizers for efficient PNM architecture and mapping co-exploration. Unlike prior frameworks [7], [42] that explore hardware or mapping in isolation with performance-centric evaluation, PNMAX uniquely co-optimizes PNM architecture and mappings, explores unconventional memory configurations (i.e., different I/O width, bank ratio, etc.) and on-chip buffer design, and exposes PPA and memory footprint Pareto tradeoffs.

Our main contributions are as follows:

- We introduce parameterized representations for both PNM architecture and tensor kernel mapping that together span a general, coupled mapping-architecture design space, enabling design points that prior frameworks cannot express.
- We develop a comprehensive, composable PPA extraction engine that enables fast system-level PPA evaluation during the design space exploration of PNM architectures. We further develop a fast modeling framework for composing the accurate system-level PPA costs and constraints for mapping-architecture co-design points.
- We develop an end-to-end framework, PNMAX, to automate the mapping-hardware co-exploration with comprehensive PPA evaluation in the complex co-design space. The PNMAX framework provides user-friendly interfaces for defining architecture design goals and target workloads.
- We use PNMAX to co-explore mappings and architectures for important ML/AI workloads. Our experiments show that PNMAX finds better mappings than prior optimization frameworks for various existing PNM architectures.
- We exploit PNMAX to path-find future designs of PNM accelerators, revealing several novel insights into mapping-architecture co-design under unconventional architectural design choices.

II. BACKGROUND AND MOTIVATION

A. Processing-near-Memory Architectures

PNM architectures enhance conventional DRAM with computation capabilities near the memory arrays, exploiting high internal bandwidth while reducing costly data movement for memory-intensive workloads [9], [10], [15], [16], [18]–[20], [44], [45]. Existing PNM architectures can be classified across four primary dimensions (Figure 1):

Memory Technology. The underlying memory standard and device technology (e.g., HBM [46], DDR4 [47]) establish the basic timing constraints and process node for area and energy calculation [48], [49]. Relevant attributes include *memory parameters* (capacity, frequency, process node, etc.) and *memory-unit configurations* (e.g., mat size, aspect ratio, row-buffer size, etc.); they dictate the data movement granularity and form the PNM system foundation.

Memory Topology. Typically, industry standards dictate the base memory organization (e.g., channels, bank groups, mats,

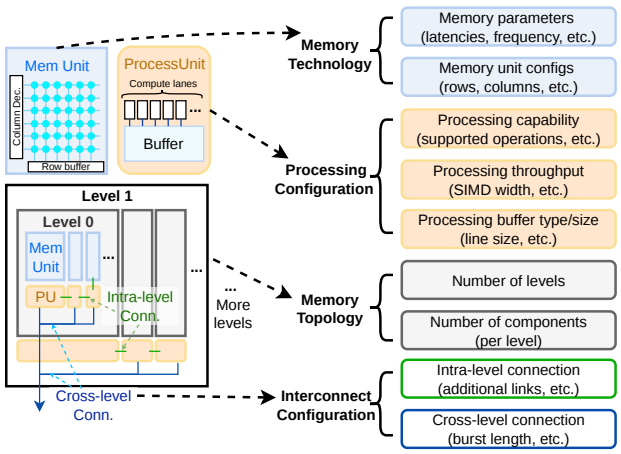


Fig. 1. Design space of PNM architectures.

etc.) and the associated hierarchy level count [9], [10], [12], [44], [50]. Designers determine the *number and specific configuration* of components (e.g., mem units, PUs) at each hierarchy level to exploit concurrency and optimize data locality. These choices significantly influence how efficiently the PNM architecture exploits available bandwidth.

Processing Configuration. PNM processing is realized by integrating specialized processing units (PUs) in the memory hierarchy; they perform word-parallel arithmetic computations while maintaining compatibility with conventional memory operations [9], [10], [17]. Processing configurations specify, for each memory hierarchy level, (1) *processing capabilities*, including supported operations (e.g., MAC, reduction tree), (2) *processing throughput*, such as compute lane count or concurrency level, and (3) *local storage type and size within PUs*. These decisions strongly impact performance attributes like concurrency, data locality, and data movement overheads.

Interconnect Configuration. The interconnect fabric defines how components within the PNM architecture communicate. (1) *Intra-level connections* facilitate data movement within a single memory level (e.g., between row buffers and PU register files) and feed local data to PUs. (2) *Cross-level connections* are inter-level links (e.g., bank group networks or through-silicon vias in HBM) that support bulk data transfer, broadcast [19], [20], [51], and streaming from external hosts to local PU buffers [9], [10].

Challenge 1: Lack of a flexible, parameterized architecture representation. To effectively explore the architectural design space and evaluate relative PPA tradeoffs, the underlying architecture representation must be flexible enough to capture diverse design dimensions and explicit hardware parameters. These parameters are essential for the downstream PPA extraction engine and mapper to identify optimal hardware-software configurations. Existing frameworks, however, either lack a generalized representation, being limited to predefined target architectures [40], or lack sufficient parametric depth and explicitness [7]. Consequently, they fail to express *processing* and *interconnect configuration* (e.g., reduction units placed at different memory hierarchy in AttAcc [10] and host to PU streaming connections in HBM-PIM [9]).

B. Mapping Tensor Programs onto PNM

Tensor kernel mapping is a relevant task [7], [40]–[42], [52] that considers two tightly coupled aspects: *operation scheduling*, determining where, when, and how operations execute; and *data layout*, specifying how tensor data is organized and accessed across memory hierarchy.

Operation Scheduling. Operation scheduling determines (1) the distribution of computations across PUs, and (2) the order of computations processed on each PU. Various representations specify tensor operation scheduling [7], [40], [42], [52]–[57]. The representation choice determines what mapping knobs are exposed to the mapper: it controls how computation is partitioned and ordered across hardware levels, and which data reuse opportunities remain visible and tunable. For instance, a nested loop representation [40], [42] explicitly exposes loop bounds and tiling factors at each memory hierarchy level as optimization variables, enabling the mapper to exploit spatial locality across iterations.

Data Layout. Data layout indicates how different tensors reside in the memory. Unlike conventional ASIC accelerators, PNM architectures have more restrictions on data layout [9], [10], [39], [40], which introduces several PNM-specific, architecture-dependent design dimensions, as shown in Figure 2 using a 1D convolution as a running example: (1) **Data residency** denotes whether the data resides within the PNM device or the host. If it resides in PNM, the accelerator must allocate the corresponding memory space; otherwise, the data should be streamed from the host during execution. (2) **Data replication strategy** indicates how data elements are replicated to align with SIMD processing required by many PNM architectures. These architecture-dependent design dimensions significantly impact the memory requirement and performance across different PNM architectures [9], [10], and are critical in software-hardware co-exploration.

Challenge 2: Lack of a generalized and explicit mapping representation. Existing PNM frameworks lack a mapping representation that explicitly covers all mapping decisions. UniNDP [7] uses PNM-oblivious lowering [43] to decompose workloads into isolated MVM sub-kernels, scoping layout decisions locally, and forfeiting inter-kernel reuse at the cost of redundant tensor duplication. Figure 2(b) shows the UniNDP operation scheduling for a conv1d kernel, where all filter elements and identical input elements across different sliding windows are duplicated without any reuse. OptiPIM [40] unifies scheduling and layout using architecture-aware nested loops. Yet, its mapping representation lacks explicit mechanisms to represent data residency (streaming) and replication (sharing). A single OptiPIM schedule can map to multiple valid data layouts with drastically different memory footprints and communication costs. Figure 2(c) is an OptiPIM operation scheduling for a conv1d kernel that can result in 2 different data layouts: Figure 2(d) stores the filter and the output in

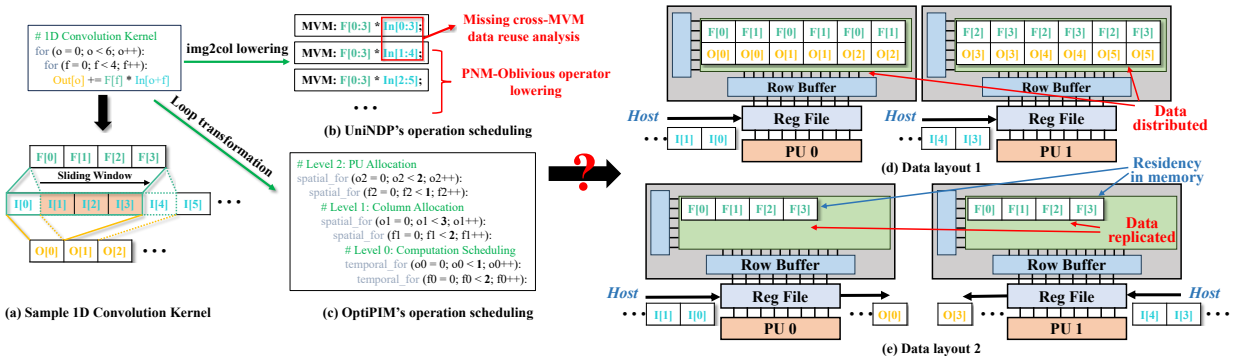


Fig. 2. (a) Sample 1D convolution. (b) UniNDP lowers the kernel into independent MVMs, so neighboring-window overlap appears only as repeated entries in the lowered inputs. (c) OptiPIM conducts loop transformations, but the same schedule can still correspond to multiple legal data layouts with different memory footprint and communication costs.

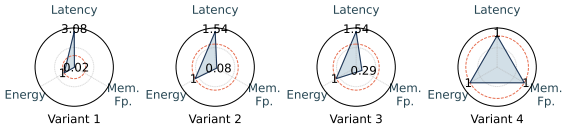


Fig. 3. Relative latency, energy, and memory footprint of best EDMP mappings normalized to best-latency mappings of Llama-3 MHA on four variants of the same PNM architecture.

the memory banks, streams inputs to the PUs, and replicates all output and filter elements according to the schedule; Figure 2(e) stores only the filter, streams the input to and output away from the PUs, and stores a full copy of the filters in all memory banks. Both frameworks handle data placement and delivery through predefined policies based on scheduling rather than tunable mapping variables, preventing systematic optimization of these decisions during DSE. An effective mapping representation must instead preserve global schedule structure while treating residency, streaming, replication, and sharing as first-class decisions.

C. Co-Exploration with PPA Tradeoffs

To design a PNM system, designers must explore the hardware architecture design space with comprehensive considerations of various metrics, however such methodology is difficult given the design of existing PNM DSE frameworks.

Challenge 3: Lack of PPA consideration in existing works. Existing PNM design frameworks [7], [40] cannot effectively support mapping-architecture co-exploration with sufficient consideration of PPA-related metrics. Existing frameworks only focus on optimizing the latency during the exploration. This one-dimensional focus overlooks complementary metrics such as energy efficiency and memory capacity utilization. Figure 3 shows the latency, energy, and memory footprint trade-offs for different mapping targets, including best-latency and best energy-delay-memory-product (EDMP), while exploring architectural design choices on UPMEM and HBM-PIM. Comparing the best-latency and the best-EDMP mappings for each architecture configuration, we find that the best-latency mapping is only slightly faster than the best EDMP mapping, but requires $> 2\times$ more memory footprint. A latency-only framework would ignore such trade-offs, which calls for a PNM DSE framework that can jointly optimize PPA and memory footprint.

III. PNMAX Overview

In this work, we propose PNMAX, a design space exploration (DSE) framework that jointly optimizes software mapping and hardware architecture for PNM-based tensor acceleration. Similar to prior tensor mapping and architecture optimization frameworks [42], [52], [54], [58], [59], PNMAX targets dense tensor workloads representable as nested affine loops with little or no control flow. This includes many dense ML and non-ML tensor computations (e.g., stencil [60]) across critical application domains targeted by various industrial and academic PNM designs [7], [9], [10], [17], [21]. To solve the aforementioned challenges in Section II, PNMAX builds on three core contributions: (1) a parameterized, graph-based architecture representation (GPAM) that models diverse PNM topologies and component configurations; (2) a flexible architecture-aware mapping representation that augments nested-loop scheduling with explicit data layout decisions to capture spatial reuse and streaming behaviors; and (3) a bottom-up PPA engine with an analytical modeling methodology that derives accurate system-level costs to evaluate performance, energy, area, and memory footprint tradeoffs. Together, these contributions enable systematic co-exploration of the highly coupled mapping-architecture design space that prior frameworks cannot express or evaluate.

Figure 4 illustrates the PNMAX framework. The process begins with a target tensor workload and an architecture template defined using our GPAM specification (Section IV), which bounds of the hardware search space. From this parameterized template, the *Architecture Generator* instantiates valid PNM configurations, while the *PPA Engine* (Section V) extracts primitive unit costs for each component, grounding design candidates in physical power and area constraints. For each

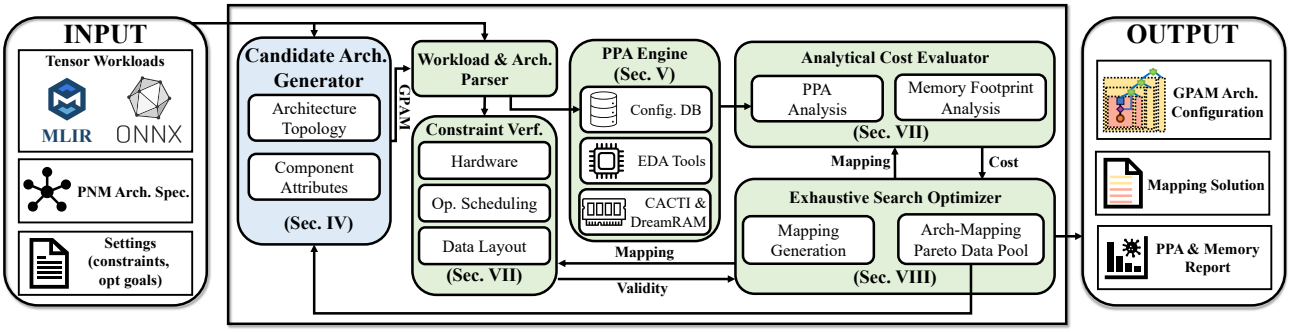


Fig. 4. Overview of the PNMAX framework.

workload-architecture pair, PNMAX formulates a comprehensive mapping space (Section VI) and evaluates candidates using our analytical cost model (Section VII). PNMAX is optimizer-agnostic and extensible to any custom or off-the-shelf search algorithm. To demonstrate the effectiveness of our co-exploration methodology, we employ time-bounded random search over the architecture design space (Section VIII), evaluating each candidate across the full workload and ranking by a user-specified aggregate objective (e.g., average latency or energy-delay product).

Framework Outputs. PNMAX bridges hardware exploration and software deployment in a single flow. It produces: (1) the optimized architecture configuration for the target workload, (2) the per-workload mapping solution (i.e., operation schedule and data layout decisions), and (3) PPA and memory footprint estimates.

IV. GPAM: Graph PNM Architecture Model

In this section, we illustrate GPAM (**G**raph-based, **P**arameterized **A**rchitecture **M**odel), a graph representation for PNM architectures. GPAM *expressively* captures diverse storage, compute, and interconnect resources, *flexibly* describes families of candidate architectures, and *explicitly* encodes detailed component-wise configurations.

A. GPAM Representation

In GPAM, nodes represent *physical resources* (e.g., DRAM banks, buffers, PUs) and edges represent *communication paths*. GPAM encodes the spatial memory hierarchy, dictating PU placement and data routing between memory and compute. Each node is annotated with microarchitectural attributes (e.g., bank/mat sizes, supported operations, SIMD width). Instantiating a GPAM graph yields a fully specified architecture instance; it dictates hardware constraints for generating legal mappings, and provides component-level parameters for the PPA engine (Section V).

Figure 5 shows representative GPAM instances for HBM-PIM [9], UPMEM [44], and AttAcc [10], together with a YAML specification format. We detail the YAML format, along with complete specifications for sample architectures, in the PNMAX repository [61]. GPAM uses four node types to represent the main resource classes in PNM: memory nodes

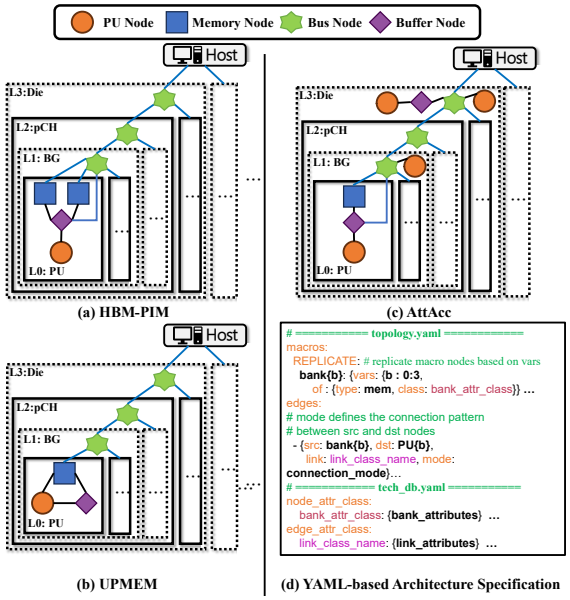


Fig. 5. Examples of GPAM representations.

(**MEM**), buffer nodes (**BUF**), processing-unit nodes (**PU**), and communication nodes (**BUS**), where **MEM** and **BUS** nodes act as boundaries between different hierarchy levels.

Memory nodes (MEM) represent persistent storage (e.g., DRAM banks). Attributes include capacity, geometry (row/-column structure), and timing parameters (e.g., latency, bandwidth), derived from *memory technology*; they directly affect data movement granularity and memory-level layout decisions.

Buffer nodes (BUF) represent explicit near-compute storage (e.g., register files, scratchpads, SRAM buffers). Their explicit modeling is important because their capacity and access characteristics directly affect data reuse, staging behavior, and mapping legality.

Processing-unit (PU) nodes represent computational resources embedded in the memory hierarchy. Their attributes specify supported operations (e.g., MAC units, adder trees) and SIMD width. This explicit modeling distinguishes different compute organizations, including specialized units (e.g.,

reduction or activation units).

Communication nodes (BUS) represent shared communication resources, such as intra-bank buses, bank-group interconnects, and TSV bundles. Their attributes describe burst size and whether a bus is an original DRAM bus or a custom interconnect, which impacts the corresponding PPA costs. Edges also carry information needed for system-level reasoning, including directionality, transfer bandwidth, and streaming support. Together with BUS nodes, they make communication structure explicit in the GPAM representation.

B. GPAM Expressiveness and Flexibility

GPAM can represent all PNM architectures supported by prior frameworks [7], [40], and extends beyond them along two axes. First, the graph topology is unconstrained: designers can freely compose node types to construct any PNM organization, including multi-level PU placement (i.e., AttAcc [10], Figure 5(c)) and custom interconnect fabrics not expressible in prior work. Second, the node-type is modular and extensible, where new resource classes can be added by defining node attributes and their interface to the PPA engine to generate the unit costs needed in the analytical model.

GPAM supports both concrete and parametric specifications through the same YAML interface. Fixing all attributes yields a single target architecture, whereas leaving selected attributes symbolic or assigning candidate ranges defines an architecture family for joint architecture-mapping exploration. These same parameters also form the interface to the PPA engine: memory geometry selects DRAM characterization points, buffer size and access width determine SRAM or RF costs, PU type and width determine post-route unit costs, and communication attributes determine interconnect cost. As a result, GPAM serves as the common interface between architecture generation, mapping, and backend PPA extraction.

V. PPA Extraction Engine

We propose a composable and flexible PPA evaluation framework that reflects how the system's latency, energy, and area vary as hardware parameters and mappings change.

PNMAX's PPA engine extracts *primitive unit costs* for each component in a GPAM instance: for **PU** nodes, operating frequency, per-operation energy, and area; for **MEM** nodes, bank-level read/write latency, energy per access, and area; for **BUF** nodes, access latency, energy, and area; for **BUS** nodes, per-transfer energy and latency as a function of burst size and topology. These primitive costs are the leaves from which the analytical cost model (Section VII) assembles system-level metrics, such as the cost of host-to-PU data streaming, the latency of a PU SIMD computation, or the energy of a cross-bank reduction, given the GPAM and the candidate mapping.

We design the engine around three properties. **Fidelity**: we select backends per component class so that costs faithfully reflect hardware behavior across the parameter space. **Reusability**: we store characterized entries in a persistent database keyed by component type and parameter configuration; subsequent design points reuse cached entries, and we dispatch only unseen configurations to the backend, amortizing cost across large explorations. **Composability**: the analytical model assembles system-level PPA from primitive entries without rerunning any backend tool.

We achieve fidelity through component-specific backends. We characterize **MEM** nodes with CACTI-3DD [48] for DDR-class organizations and DreamRAM [49] for HBM-class organizations, capturing latency, energy, and area as a function of bank/mat organization, aspect ratio, and burst length. We characterize **BUF** nodes with CACTI-7.0 [62]. For **PU** nodes, we use a parameterized RTL generation flow: we generate SIMD floating-point units with FPnew [63] and specialized units from hardware templates in XLS [64]; user-provided RTL or XLS descriptions can be incorporated analogously. We perform synthesis with Genus v25.10 [65] and place-and-route with Innovus v25.10 [66] using the Nangate45 45 nm (v1.2) technology library [67]. To approximate logic embedded in a DRAM process, we constrain physical design to a DRAM-compatible three-metal-layer routing stack [44]. Since the open-source logic library and target memory process are not technology-matched, we scale post-route results using published cross-node factors to a 22nm technology node [68] (divide by 4.54 for area, 1.30 for delay, and 2.1 for energy). We estimate the PPA impacts of custom interconnects using CACTI's analytical wire models [62], parameterized by geometry (width and spacing) and topology. Further details are provided in the PNMAX repository [61].

Taking HBM-PIM (Table III) as an example, PNMAX maps each GPAM component to its corresponding characterization backend. The metadata of HBM-PIM's PU node, including the data format (FP16), the operation group (ADDMUL), vector mode (enabled), number of SIMD lanes (16), and number of pipeline stages (sweep across 1 to 3), is used to configure FPnew [63] to generate the RTL code. The generated RTL is then synthesized and placed and routed using the PU-specified clock period, 70% floorplan utilization, and a three-metal-layer routing constraint. After technology scaling [68], the resulting area and per-operation energy are stored as PU unit costs. For the memory subsystem, the GPAM topology, including the channel, pseudo-channel, and bank-group organization, together with MEM-node parameters such as row size and mat geometry and BUS-node parameters such as burst length, is translated into DreamRAM [49] inputs to obtain DRAM timing and energy costs, including tRP, tRCD, and tCL. Similarly, each BUF node's capacity and I/O width are passed to CACTI [62] to obtain its area and access energy. These component-level costs are then composed by the analytical model (Section VII).

PNMAX evaluates system-level PPA through a modular analytical model (Section VII). The component-extraction engine composes outputs from established, independently verified tools, including CACTI for DDR4, DreamRAM for HBM2, and standard RTL place-and-route flows for logic, each rigorously validated. PNMAX integrates their unit costs into our component library, and like any analytical model, calibrates against every golden reference of the PNM baselines.

VI. PNM Mapping Representation

We propose an architecture-aware mapping representation with two complementary components: (1) a *hierarchy-aware* nested-loop schedule that captures computation partitioning and execution order, and (2) *explicit* layout pragmas that capture PNM-specific data placement choices. This representation removes the ambiguity of scheduling-centric layout representations (Figure 2), where the same schedule can induce multiple legal layouts with different communication costs, memory footprints, and reuse behaviors.

A. Hierarchical Scheduling Representation

Like prior works [40], [42], PNMAX uses the nested loop as the basic representation to capture temporal and spatial characteristics for scheduling tensor-kernel operations. While prior work only represents the computation distribution across PUs [40], our representation explicitly tiles the loop structure at each memory hierarchy level, enabling fine-grained hierarchical layout decisions. Specifically, the schedule is expressed as a nested loop (Figure 6(b)) whose structure is aligned with the target GPAM instance (Figure 6(a)). Outer loop levels (level-2 and higher) map tensor computations across the memory hierarchy, determining which hardware instances participate and how computation is partitioned across channels, bank groups, and banks. Inner loop levels define per-PU execution, namely the temporal ordering of computation steps (level-1) and intra-PU parallelism such as SIMD lanes [9], [10] or parallel tasklets [44] (level-0). Together, these loops determine the active resources, execution order, and operations issued at each timestep. We denote the data elements of input and output tensors required for each iteration of the innermost loop (level-0) as tensor *tiles*.

Figure 6 illustrates a 1D-convolution kernel mapping onto a four-level GPAM architecture. In this example, bounds for Levels 5, 4, and 3 are set to 1, and all spatial partitioning is specified at the bank level (Level 2), with two PUs each assigned to one memory bank. The innermost loops (Levels 1 and 0) dictate that each PU loads one filter (F) *tile* and one input (In) *tile* to generate one output (Out) *tile* per timestep. These multi-dimensional tensor *tiles* serve as the fundamental unit of data movement and placement, and Figure 6(c) visualizes how they are physically laid out in memory, jointly controlled by the loop structure and layout pragmas.

PNMAX assumes the host handles reduction (if not handled by reduction units in PNM architectures such as AttAcc [10]) and data transfer, which are executed sequentially. Take multi-head attention (MHA) for instance: the host streams inputs to the PUs, handles reduction and activation functions, and transfers inter-PU data. As our cost model (Section VII-B) captures the reduction and data transfer through host, the coordination can hence be effectively optimized in PNMAX.

B. Explicit Data Layout Pragmas

The hierarchical schedule alone does not uniquely determine the final tensor layout in PNM without an implicitly assumed layout policy. As shown in Figure 2(c-e), the same nested loop schedule can describe different concrete data layouts: tensors can reside in memory or be streamed into local storage, and tensor tiles can be replicated or shared across hierarchy levels. These choices directly affect memory footprint, communication volume, and locality. To make these layout choices into first-class mapping decisions, PNMAX augments the nested loop with explicit layout pragmas.

Residency pragma: $\mathcal{D}_{\mathrm{v}}^{\mathrm{host}}$. This pragma controls how tensor v is supplied to the processing unit, with $\mathcal{D}_{\mathrm{v}}^{\mathrm{host}} \in \{0, 1\}$: (1) $\mathcal{D}_{\mathrm{v}}^{\mathrm{host}} = 1$: Tensor elements reside in the host and are streamed directly into PU-local buffers for immediate computation, bypassing explicit storage in memory banks. Some PNM architectures, such as AttAcc [10] and HBM-PIM [9], support this mode (Figure 2(d) and (e)). (2) $\mathcal{D}_{\mathrm{v}}^{\mathrm{host}} = 0$: Tensor elements reside in memory units and are explicitly loaded into PU-local buffers or registers when required.

Sharing pragma: $\mathcal{D}_{\mathrm{l,v}}^{\mathrm{share}}$. This pragma explicitly governs tensor replication across the memory hierarchy, thus balancing memory footprint and data movement overhead. Here, l denotes the loop level index in the transformed nested loop, and the pragma setting $\mathcal{D}_{\mathrm{l,v}}^{\mathrm{share}} \in \{0, 1\}$ specifies the replication policy for different tensors at different loop levels. If sharing is enabled at one level, replicated tensor tiles will not be stored in the corresponding memory level. Existing approaches often rely on fixed strategies: OptiPIM [40] replicates tensor elements across all banks (Figure 6(c)), whereas UniNDP [7] maintains only a single global tensor copy per MVM/MM sub-kernel. In contrast, our pragma provides the flexibility to define various sharing strategies. Figure 2(d) and (e) show the effect of enabling sharing at different loop levels.

C. Expressiveness of PNMAX's Mapping

A complete mapping representation must express: (1) how data moves to and from the PU, (2) where and how computation is partitioned and ordered, and (3) how data is placed across the memory hierarchy. PNMAX addresses each through an orthogonal component: the $\mathcal{D}_{\mathrm{v}}^{\mathrm{host}}$ pragma governs data movement to and from the PU; the nested loop schedule defines the spatial and temporal computation coordinates; the $\mathcal{D}_{\mathrm{l,v}}^{\mathrm{share}}$ pragma controls tile replication and memory footprint. The combination of these three orthogonal components uniquely determines every load, store, streaming transfer, and replication, eliminating the hardcoded backend assumptions of prior frameworks and ensuring that every valid layout-scheduling configuration can be systematically explored.

VII. PNMAX Cost Modeling and Optimization

As discussed in Section II, mapping-architecture co-exploration requires a cost model that is fast, accurate, and sufficiently general to evaluate a large space of architectural configurations and mappings. PNMAX introduces an analytical model that evaluates four metrics: *memory footprint, execution latency, total energy*, and *total area*. Figure 7 shows the calculation of the metrics. An instantiated GPAM (Section IV, Figure 7(a)), an architecture-aware schedule with layout pragmas (Section VI, Figure 7(b)), and per-component

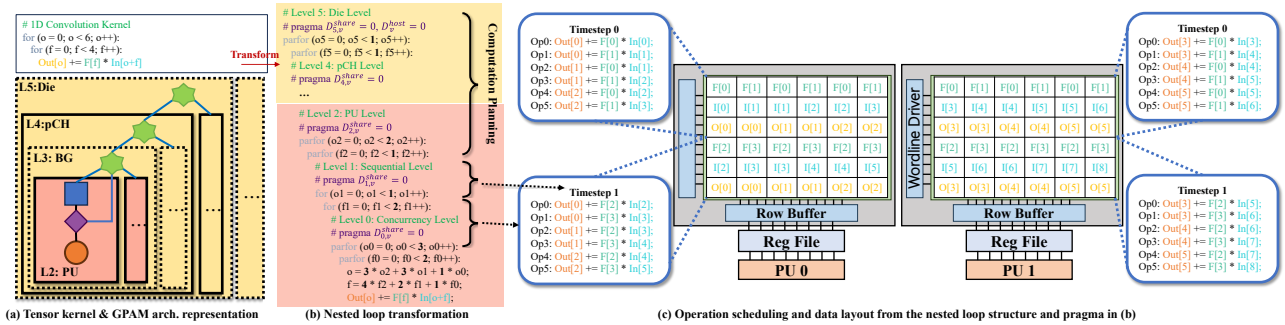


Fig. 6. (a) 1D Conv and PNM arch in GPAM. (b) Nested-loop-based mappings and layout pragmas. (c) Data layout from (b).

PPA values from the extraction engine (Section V) are needed to calculate the four metrics in Figure 7 (c) and (d).

Our analytical model achieves the required speed, accuracy, and generality by composing system-level costs from three sources, summarized in Table II: (1) per-component PPA unit costs produced by the PPA engine and stored in the GPAM representation (Figure 7(a) ⓐ), (2) architecture parameters specified by the user in GPAM, such as topology, operating frequency, and bank organization (Figure 7(a) ⓑ), and (3) quantities derived from the selected mapping, such as tile counts, timestep counts, and data movement volumes (Figure 7(b) ⓒⓓⓔ). This decomposition captures architectural heterogeneity and mapping-dependent behavior while keeping evaluation tractable. To balance fidelity and efficiency, PNMAX uses tensor tile as the basic unit for execution and data placement, so latency, energy, and memory footprint can be aggregated systematically from tile-level operations. Hardware-specific constraints are enforced throughout to ensure only legal mapping-architecture combinations are evaluated. Together, the analytical cost model and hardware constraints form a constrained optimization problem, where the cost model defines the objective and the constraints bound the feasible design space, enabling any optimizer to efficiently search for optimal mapping-architecture configurations.

A. Memory Footprint

As shown in Figure 7(c), memory footprint is determined by the set of *stored* tensor tiles and their placement across the MEM nodes of GPAM (controlled by the schedule and layout pragmas). Streamed tensors do not occupy persistent bank-level storage, whereas stored tensors must be allocated to the corresponding MEM nodes according to the mapping. Let $\mathbf{b}_v^{\text{tile}}$ denote the number of bytes in one tile of tensor v, determined by the level-0 (PU concurrency spatial `for`) loop bounds; and let $\mathbf{N}_v^{\text{tiles}}$ denote the number of tiles of v placed in MEM node m, determined by the level-1 (temporal `for`) loop bounds and the sharing policy. The footprint of m is then

$$\mathcal{MF}_m = \sum_{v \in \mathbf{S}(m)} \left\lceil \frac{\mathbf{N}_v^{\text{tiles}} \cdot \mathbf{b}_v^{\text{tile}}}{\mathbb{B}_m} \right\rceil \cdot \mathbb{B}_m,$$

where $\mathbf{S}(m)$ is the set of tensors stored in node m, and $\mathbb{B}_m$ is the allocation granularity of that node (e.g., DRAM-row bytes). The total memory footprint across all MEM nodes is $\mathcal{MF} = \sum_{m \in \text{MEM}} \mathcal{MF}_m$. This formulation explicitly models the data layout in DRAM rows under the effect of streaming and sharing decisions, and therefore captures the footprint, which cannot be determined from the scheduling alone. In Figure 6(c), the memory footprint of the layout is $\mathcal{MF} = (4+4+4) \cdot \mathbb{B}_m = 12 \cdot \mathbb{B}_m$; each tensor takes two rows in each PU, resulting in 4 rows per tensor in total.

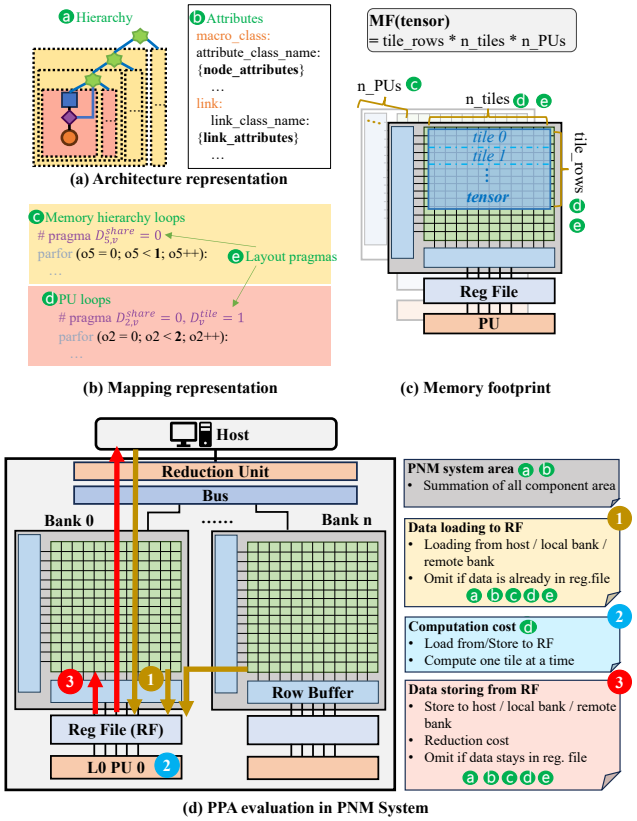


Fig. 7. Analytical cost model: deriving (c) memory footprint and (d) PNM-system performance, energy, and area from components in (a) the GPAM architecture representation and (b) the mapping representation.

TABLE II
ANALYTICAL COST MODEL NOTATION. SYMBOL STYLE REFLECTS ORIGIN: $\mathbb{X}$ = USER-SPECIFIED ARCHITECTURE PARAMETER (GPAM); $t_{\mathrm{x}}/e_{\mathrm{x}}/a_{\mathrm{x}}$ = PPA ENGINE OUTPUT STORED IN GPAM; $\mathbf{X}$ = MAPPING-DERIVED QUANTITY; $\mathcal{X}$ = SYSTEM-LEVEL COST BY PNMAX.

Symbol	Unit	Description
User-Specified Architecture Parameters (GPAM)		
$\mathbb{B}_{\mathrm{m}}$	bits	Allocation granularity of MEM node m (e.g., DRAM row width).
$\mathbb{N}^{\mathrm{PU}}$	—	Number of processing units.
$\mathbb{T}_{\mathrm{topo}}$	—	Memory topology (banks, ranks, channels).
$\mathbb{G}_{\mathrm{x}}$	—	Set of hardware groups involved in operation x.
$\mathbb{P}_{g_{\mathrm{x}}}$	—	Set of operation x paths in a hardware group g_{x}.
$\mathbb{U}_{\mathrm{spec}}$	—	Set of specialized units, such as reduction or activation.
PPA Engine Outputs (stored in GPAM)		
t_{x}	cycles	Latency unit cost for component or operation x (e.g., t_{RP}: DRAM row-precharge latency).
e_{x}	pJ	Energy unit cost for component or operation x (e.g., e_{Buf}: PU buffer access energy).
a_{x}	mm^2	Area of component x (e.g., a_{PU}: area of one processing unit).
Mapping-Derived Quantities		
$\mathbf{N}^{\mathrm{tiles}}_{v,m}$	—	Number of tiles of tensor v placed in MEM node m.
$\mathbf{b}^{\mathrm{tile}}_{v}$	bits	Size of one tile of tensor v.
$\mathbf{S}(m)$	—	Set of tensors stored in MEM node m.
$\mathbf{N}^{\mathrm{steps}}$	—	Number of sequential PU steps (timesteps).
$\mathbf{N}^{\mathrm{trans}}_{p}$	—	Number of transfer operations on data path p.
$\mathbf{N}^{\mathrm{PU}}$	—	Number of PUs used by the mapping, $\leq \mathbb{N}^{\mathrm{PU}}$.
$\mathbf{R}_{\mathrm{x}}$	—	Ratio of PU data from source x to the total number of PU steps. $\sum_{\mathrm{x}} R_{\mathrm{x}} = 1$.
System-Level Costs (computed by PNMAX)		
$\mathcal{T}$	cycles	Total execution latency ($\mathcal{T}_{\mathrm{input}}+\mathcal{T}_{\mathrm{PU}}+\mathcal{T}_{\mathrm{store}}$).
$\mathcal{T}_{\mathrm{input}}$	cycles	Total execution latency for delivering operand tiles to near-PU storage.
$\mathcal{T}_{\mathrm{PU}}$	cycles	Total execution latency for PU steps.
$\mathcal{T}_{\mathrm{store}}$	cycles	Total execution latency for output storing and reduction.
$\mathcal{E}$	pJ	Total execution energy ($\mathcal{E}_{\mathrm{input}}+\mathcal{E}_{\mathrm{PU}}+\mathcal{E}_{\mathrm{store}}$).
$\mathcal{E}_{\mathrm{input}}$	pJ	Total execution energy for delivering operand tiles to near-PU storage.
$\mathcal{E}_{\mathrm{PU}}$	pJ	Total execution energy for PU steps.
$\mathcal{E}_{\mathrm{store}}$	pJ	Total execution energy for output storing and reduction.
$\mathcal{A}$	mm^2	Total chip area.
$\mathcal{MF}$	MB	Peak memory footprint.

B. Execution Latency and Energy

Our analytical model calculates latency and energy by decomposing hardware execution into three phases, shown on the PNM system in Figure 7(d): ❶ input loading, ❷ PU execution (including arithmetic, local data supply and storage, row activations, and remote fetch movement), and ❸ output storage, while accounting for multi-level reduction, interconnect, and tensor buffering.

a) External input supply: $\mathcal{T}_{\mathrm{input}}$ and $\mathcal{E}_{\mathrm{input}}$ account for delivering operand tiles to near-PU storage before each PU step. Let $\mathbb{P}_{g_{\mathrm{in}}}$ denote the set of such transfer paths in a hardware group, and $\mathbb{G}_{\mathrm{input}}$ the set of hardware groups involved in the transfers. A hardware group is the set of homogeneous components that can perform the same types of operation in parallel, e.g., host streaming to several PUs in parallel. Then

$$\mathcal{T}_{\mathrm{input}} = \max_{g_{\mathrm{in}}\in\mathbb{G}_{\mathrm{input}}} \sum_{p_{\mathrm{in}}\in\mathbb{P}_{g_{\mathrm{in}}}} \mathbf{N}^{\mathrm{trans}}_{p_{\mathrm{in}}}\, t_{\mathrm{p_{in}}},$$

$$\mathcal{E}_{\mathrm{input}} = \sum_{g_{\mathrm{in}}\in\mathbb{G}_{\mathrm{input}}} \sum_{p_{\mathrm{in}}\in\mathbb{P}_{g_{\mathrm{in}}}} \mathbf{N}^{\mathrm{trans}}_{p_{\mathrm{in}}}\, e_{\mathrm{p_{in}}},$$

where $\mathrm{N}^{\mathrm{trans}}_{p_{\mathrm{in}}}$ is the number of input transfer operations on path p determined by the mapping. The unit cost $(t_{\mathrm{p}}, e_{\mathrm{p}})$ is selected according to the source and path type, e.g., host-to-PU streaming cost.

b) PU execution and reduction: $\mathcal{T}_{\mathrm{PU}}$ and $\mathcal{E}_{\mathrm{PU}}$ account for PU steps and any specialized processing stages activated by the architecture. Before the arithmetic operations, the PUs need to load data internally, e.g., reuse from a local buffer, local bank-level storage, or fetch remotely (e.g., inter-bank, inter-channel). Let $\mathbf{N}^{\mathrm{steps}}$ be the number of PU steps implied by the sequential schedule, $\mathbf{R}_{\mathrm{buf}}$, $\mathbf{R}_{\mathrm{bk}}$, $\mathbf{R}_{\mathrm{fe}}$ the ratio of local buffer reuse, bank loads, and remote fetches respectively, and $(t_{\mathrm{ari}}, e_{\mathrm{ari}})$, $(t_{\mathrm{buf}}, e_{\mathrm{buf}})$, $(t_{\mathrm{bk}}, e_{\mathrm{bk}})$, $(t_{\mathrm{fe}}, e_{\mathrm{fe}})$ the unit latency and unit energy of one arithmetic operation, local buffer reuse, bank load, and remote fetch respectively. If specialized stages (e.g., reduction units) exist, their cost is also accumulated:

$$\mathcal{T}_{\mathrm{PU}} = \mathbf{N}^{\mathrm{steps}} \cdot (t_{\mathrm{ari}} + \mathbf{R}_{\mathrm{buf}} t_{\mathrm{buf}} + \mathbf{R}_{\mathrm{bk}} t_{\mathrm{bk}} + \mathbf{R}_{\mathrm{fe}} t_{\mathrm{fe}}) + \sum_{r\in\mathbb{U}_{\mathrm{spec}}} \mathbf{N}_r\, t_{\mathrm{r}},$$

$$\mathcal{E}_{\mathrm{PU}} = \mathbf{N}^{\mathrm{PU}} \cdot [\mathbf{N}^{\mathrm{steps}} \cdot (e_{\mathrm{ari}} + \mathbf{R}_{\mathrm{buf}} e_{\mathrm{buf}} + \mathbf{R}_{\mathrm{bk}} e_{\mathrm{bk}} + \mathbf{R}_{\mathrm{fe}} e_{\mathrm{fe}}) + \sum_{r\in\mathbb{U}_{\mathrm{spec}}} \mathbf{N}_r\, e_{\mathrm{r}}],$$

where $\mathbb{U}_{\mathrm{spec}}$ is the set of specialized units, such as reduction or activation, $\mathbf{N}^{\mathrm{PU}} \leq \mathbb{N}^{\mathrm{PU}}$ is the number of PUs used by the mapping, and $\mathbf{N}_r$ is the number of invocations of stage r. Note that $\mathbf{R}_{\mathrm{buf}} + \mathbf{R}_{\mathrm{bk}} + \mathbf{R}_{\mathrm{fe}} = 1$. $\mathbf{R}_{\mathrm{buf}}$ and $\mathbf{R}_{\mathrm{bk}}$ are calculated as the hit and miss rate when the near-PU BUF node is modeled as a cache or RF. We derive hit/miss rates analytically from the sequential schedule and buffer capacity using fully associative Cache Miss Equations (CME) [69] under an LRU policy. For remote fetching paths triggered by tensor sharing, we calculate $\mathbf{R}_{\mathrm{fe}}$ from the difference between the number of stored tensor tiles before and after system-level sharing is on, accurately capturing sharing reuse without cycle-accurate simulation.

c) Output storage and hierarchical reduction: After computation, outputs or partial sums must be written back to their destination. If reduction units are placed at one or more hierarchy levels, partial sums are first combined locally and then forwarded upward; otherwise, all partial sums are propagated to the next storage level or the host. Let $\mathbb{P}_{g_{\mathrm{st}}}$ denote the set of resulting store paths in a hardware group, and $\mathbb{G}_{\mathrm{store}}$ denote the set of hardware groups involved in the store and reduction. Then

$$\mathcal{T}_{\mathrm{store}} = \max_{g_{\mathrm{st}}\in\mathbb{G}_{\mathrm{store}}} \sum_{p_{\mathrm{st}}\in\mathbb{P}_{g_{\mathrm{st}}}} \mathbf{N}^{\mathrm{trans}}_{p_{\mathrm{st}}}\, t_{\mathrm{p_{st}}},$$

$$\mathcal{E}_{\mathrm{store}} = \sum_{g_{\mathrm{st}}\in\mathbb{G}_{\mathrm{store}}} \sum_{p_{\mathrm{st}}\in\mathbb{P}_{g_{\mathrm{st}}}} \mathbf{N}^{\mathrm{trans}}_{p_{\mathrm{st}}}\, e_{\mathrm{p_{st}}},$$

where $\mathrm{N}^{\mathrm{trans}}_{p_{\mathrm{st}}}$ is the number of store and reduction operations on path p determined by the mapping. The number of oper-

TABLE III
BASELINE ARCHITECTURE SETTINGS IN THE EXPERIMENTS.

	HBM-PIM [9]	AttAcc [10]	UPMEM [14]
PU count (16-bit)	512 × 16	256 × 16	2560
Register file / PU	512B	512B	32KB
Row Size	1kB	1kB	1kB
Bank capacity	16MB	16MB	32MB
Core frequency	300MHz	666MHz	350MHz
Burst Length	4	4	4
Bank group size	4	4	4
Technology	HBM2 [70]	HBM2 [70]	DDR4 [71]

ations involved in each reduction stage should decrease with the reduced output elements. This multi-stage reduction setting enables PNMAX to consider the reduction unit placement (e.g., AttAcc [10]) specified in GPAM, the mapping of partial sum reduction, and the PNM-host coordination simultaneously. The host is modeled as the last stage of the process if involved (e.g., HBM-PIM [9]). The cost model naturally captures the tradeoff between reducing communication volume earlier in the hierarchy and paying the additional cost of the corresponding reduction stages, including the coordination with host. Finally, for a given mapping, the total execution costs are the sum of these phases:

$$\mathcal{T} = \mathcal{T}_{\text{input}} + \mathcal{T}_{\text{PU}} + \mathcal{T}_{\text{store}}, \qquad \mathcal{E} = \mathcal{E}_{\text{input}} + \mathcal{E}_{\text{PU}} + \mathcal{E}_{\text{store}}.$$

C. Total Area

The PNM architecture's area is calculated as the sum of the DRAM memory system area and the area consumed by the PU and its local buffers based on the instantiated GPAM, (Figure 7 (d)): $\mathcal{A}_{\text{PNM}} = a_{\text{DRAM}} + N_{\text{PU}} \cdot (a_{\text{PU}} + a_{\text{BUF}})$, where N_{PU} is the number of PUs, and a_c is the area of component c in the GPAM-specified architecture.

By composing extracted primitive costs with explicit mapping and architecture information, our proposed model enables rapid system-level evaluation of latency, energy, area, and memory footprint without architecture-specific simulators.

D. Scheduling and Layout Constraints

A set of scheduling and layout pragmas is valid on a given architecture only if it satisfies three constraints: (1) nested loop transformation constraints, requiring the product of a loop variable across all levels to equal the original loop variable, (2) operation scheduling constraints, forcing the product of all loop variables in each spatial level to fit within the level's capacity limit, which covers PU concurrency and the row and bank capacities set by level-1 loop bounds, and (3) streaming constraints, allowing tensor streaming only if a direct connection exists between the bus node and the near-PU buffer node. The first two constraints follow OptiPIM [40], while the third is enabled by the GPAM representation.

VIII. EXPERIMENTAL SETUP

We implement PNMAX as a 23,278-line Python prototype, comprising an architecture/workload parser, GPAM and mapping representation generator, PPA extraction engine, analytical cost model, architecture-mapping space explorer, end-to-end evaluation pipelines, and simulator validation interfaces. Experiments are done on a 64-core AMD Threadripper 9985WX machine with 768 GB memory.

Workloads. We evaluate four types of tensor kernels as workloads: 2D convolution (Conv2D), fully connected layers (FC), batched fully connected layers (Batched-FC), and Multi-Head Attention (MHA). We choose Conv2D Kernels from VGG16 [72], ResNet50 [73], AlexNet [1], FC kernels from BERT [2], ResNet50 [73], as well as batched-FC and MHA kernels from BERT [2] and Llama-3 8B [3]. All kernels use 16-bit data precision, with dimensions specified in their model designs. We also conduct end-to-end evaluations on LLaMA-3 8B [3], GPT-2-medium [74], and OPT-2.7B [75].

Baseline Architectures. Table III summarizes the default configurations of the evaluated PNM architectures. For **AttAcc** [10], we use the simulator released with the original work. For **UPMEM** [14] and **HBM-PIM** [9], we rely on the simulator from UniNDP [7] with the same architecture configurations. Specifically, HBM-PIM has two banks connected to one PU (Figure 5(a)), which loads from the two banks interleavingly. In this case, we model the HBM-PIM row size as doubled the original size to reflect the interleaving behavior. The top-level architectural knobs are interpreted into tool-specific parameters for Cacti-7.0 [62] and DreamRAM [49].

PPA Evaluation is discussed in Section V.

Mapping DSE Implementation. We perform the mapping DSE via a time-bounded random search (1 hour per architecture configuration using 128 threads) to explore and evaluate only candidate mappings that honor all architectural and workload constraints.

Baseline Mapping Strategy. We select baseline mapping strategies per architecture. For HBM-PIM, we use UniNDP and OptiPIM [40]. For UPMEM, we use UniNDP and CINM [76]. For AttAcc, we use the mapping strategy reported in the original work [10]. We use UniNDP for both HBM-PIM and UPMEM because we validate our model with the UniNDP simulator, while OptiPIM and CINM each support only one of the two architectures respectively.

IX. EVALUATION/CASE STUDY

A. Validation

To validate the correlation of latency calculations (in clock cycles) between our analytical model and UniNDP's simulator, we randomly generate 2000 valid mappings from all selected workloads for HBM-PIM [9] and UPMEM [44], and 1013 mappings for AttAcc [10] due to its more restricted mapping space. We also validate the PNMAX energy against AttAcc, the only simulator that reports energy. Figure 8 shows the scatter plots for latency and energy validation, respectively. The simulator and PNMAX runtimes, as well as the squared Pearson correlation coefficients, are shown in Table IV: for all architectures, PNMAX produces accurate results, with an average speedup of 20×. Furthermore, as illustrated in Figure 9, the PNMAX evaluation runtime/memory complexity

TABLE IV
WALL-TIME COMPARISON BETWEEN SIMULATORS [7], [10] AND PNMAX. VALUES ARE SUMMED OVER ALL WORKLOADS.

	HBM-PIM [9]	UPMEM [14]	AttAcc [10]
Simulator Wall-time	6226 (s)	4211 (s)	294 (s)
PNMAX Wall-time	194 (s)	165 (s)	119 (s)
Speedup	32.1×	25.5×	2.47×
Latency Correlation (R^2)	0.9984	0.9764	0.9967
Energy Correlation (R^2)	—	—	0.9871

is insensitive to the number of tiles for different mappings, showing the scalability of the framework. The upper cluster of peak memory usage is formed by conv2d layers. We evaluate PNMAX area for AttAcc, where PUs and buffers account for 7.85% DRAM area overhead, compared with 10.84% reported in the original literature. The difference mainly comes from the PU implementation, which is unavailable for AttAcc.

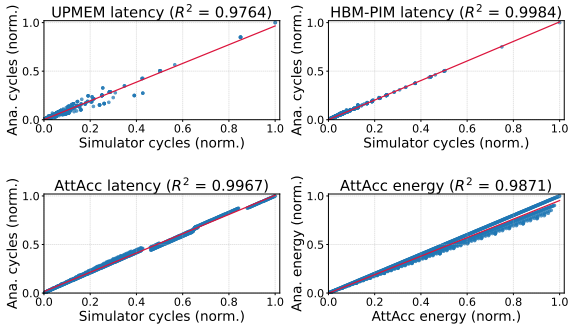


Fig. 8. Latency/energy validation: baseline simulators vs. PNMAX.

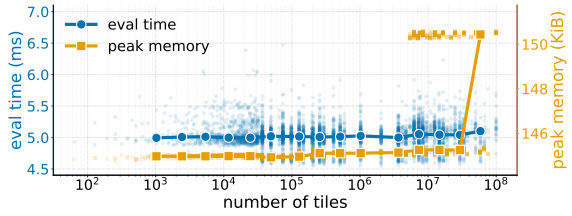


Fig. 9. Distribution of runtime/peak memory for evaluating one mapping with PNMAX, over all mappings in the validation set. Lines are binned medians.

B. Mapping DSE

1) Layout Decisions: We first use PNMAX to study how data layout decisions affect the mapping trade-off of memory footprint and latency on the baseline architectures (Table III). Starting from a restricted space with only level-0 sharing enabled and tensor buffering disabled, we progressively enable additional layout pragmas and rerun the mapping DSE. Figure 10 illustrates the resulting Pareto fronts compared against UniNDP, OptiPIM, and CINM baselines. As more pragmas are enabled, the Pareto frontier consistently improves: sharing pragmas reduce memory footprint with limited latency penalty, while tensor buffering consistently improves the overall latency by reducing data movement overhead. Furthermore, the Pareto fronts found by PNMAX dominate the UniNDP, OptiPIM, and CINM baselines (×, star, and diamond in the figures), as all three baselines tend to duplicate data needed by different MVM/MM sub-kernels without considering it. Also, UniNDP does not analyze cross-kernel data reuse. PNMAX, in contrast, can reason about tensor sharing and streaming that the baselines cannot reason about. We could add other similar features just as easily (as a pragma) and analogously find Pareto-optimal mappings for any other architecture described in GPAM. By exposing data-layout choices as composable pragmas, PNMAX explores a broader space and thus uncovers Pareto points that the baselines cannot find.

2) Tensor Streaming: To understand the role of streaming, we use PNMAX to compare the best-latency mappings with and without tensor streaming enabled, keeping the rest of the exploration flow unchanged. As shown in Figure 11, disabling streaming induces a significantly larger memory footprint, as previously streamed tensor tiles must now be stored in memory. Yet, the latency impact is architecture-dependent: UPMEM incurs a substantially higher streaming overhead relative to bank-local access than HBM-PIM, making the benefits of streaming lower than in HBM-PIM. Furthermore, UPMEM shows a less evident latency improvement when the buffer is enabled. **Insight: different architectural features can have different costs and benefits when integrated into different baseline architectures, which points to the need of an explorer like PNMAX that can systematically evaluate these effects in *any* architecture and for *any* feature.**

3) Tensor Sharing: To assess the impact of tensor sharing decisions, Figure 12 shows the latency breakdown of all mapping results (Figure 10) on HBM-PIM under the four sharing and buffer configurations as in the previous experiments. We observe that sharing mainly benefits the latency by reducing the output size and thus output storage cost. **Insight: reducing memory footprint can also lower the latency by reducing the data transmission requirement.**

Also, adding the buffer either saves streaming cost by reducing input access (streaming) or bank loading cost by reducing filter access (load from bank to PU) for BERT layer. **Insight: latency benefits the most when the buffer is mostly distributed to one specific tensor rather than across different tensors.**

Furthermore, system-level sharing does not often benefit best latency or best memory footprint in the trade-offs, as neither mapping induces inter-PU data transfer. Best latency mappings induce massive duplication to ensure locality, while best memory footprint mappings often have small tile sizes, fitting the execution locally and compromising parallelism. Section IX-C3 illustrates more on this effect. **Insight: system-level sharing provides better trade-off options between latency and memory footprint, rather than being optimal for either one of the metrics.**

4) DSE Search Process: Figure 13 shows how the best found mapping costs evolve with exploration time for

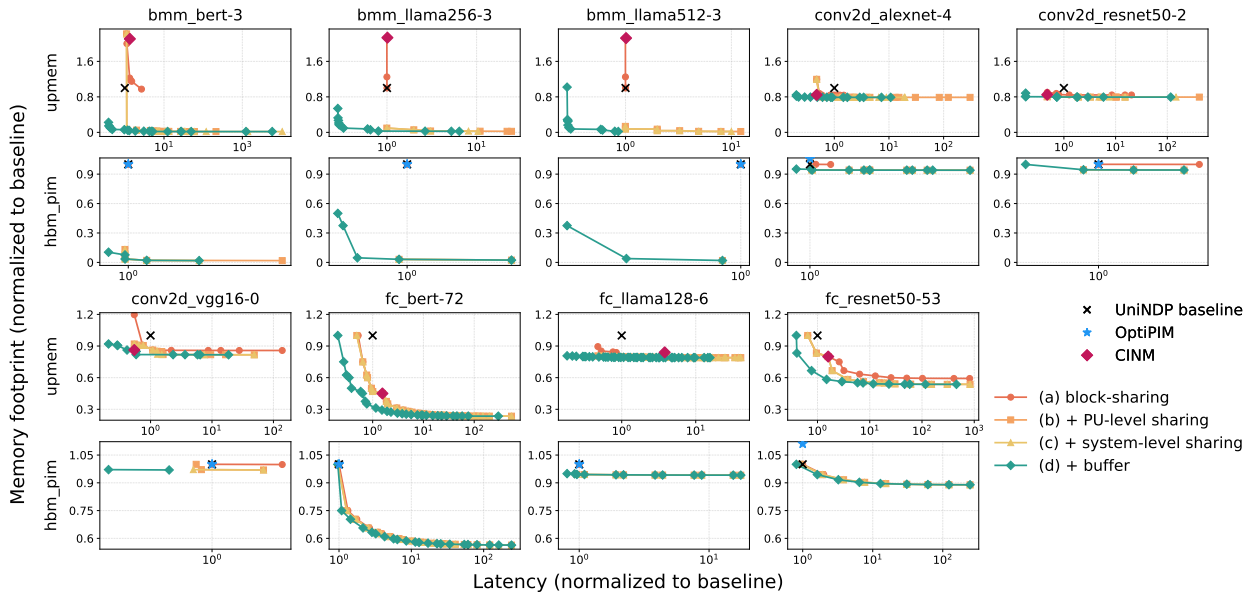


Fig. 10. Latency-memory footprint Pareto frontiers obtained by the mapping DSE across nine workloads and two architectures (UPMEM and HBM-PIM). Each subplot shows design points normalized to its corresponding UniNDP baseline, marked by ×. Also, the star and diamond mark OptiPIM and CINM baselines respectively. Points closer to the lower-left indicate simultaneous latency and memory footprint reductions. Different Pareto fronts indicate mapping spaces of: (a) Level-0 sharing only, (b) PU-level sharing enabled, (c) system-level sharing enabled, and (d) buffer-enabled configurations, each added incrementally. For example, in the top-left panel (bmm_bert-3 on UPMEM) every block-sharing-only mapping (a) lies on or above the baselines, whereas adding all pragmas (d) finds a mapping with 0.38× latency and 0.23× footprint compared with UniNDP.

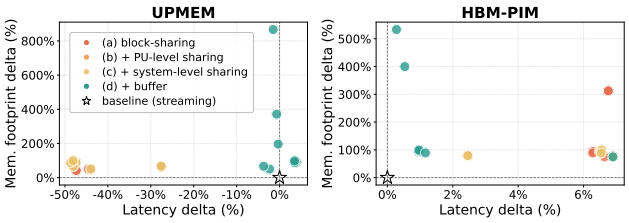


Fig. 11. Latency-memory tradeoff when disabling streaming. Each point corresponds to the best-latency mapping in each Pareto front in Figure 10, plotted as percentage change in latency and memory footprint relative to the streaming-enabled mappings (the origin point).

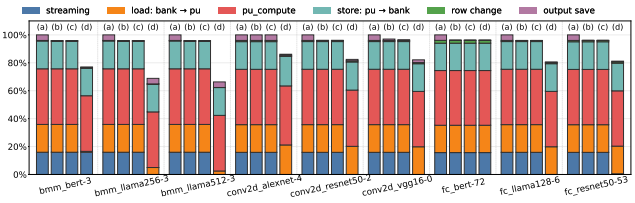


Fig. 12. Lat. breakdown of the best-latency HBM-PIM mapping for each workload and config in Figure 10. Each bar is normalized to the latency of config (a) for the same workload.

llama512-3 on HBM-PIM, with all three costs normalised to the UniNDP baseline, as an example to illustrate the DSE search process. All three metrics improve significantly early on (initial values on the y-axis in Figure 13). Then the metrics

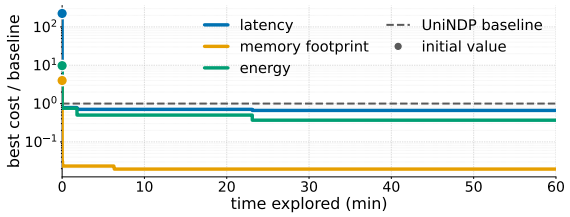


Fig. 13. Trajectory of best latency, memory footprint, and energy during the Pareto search for llama512-3 on HBM-PIM.

gradually improve at a slower pace due to slower discovery of better valid mappings. The process converges after 23 minutes at local minimum points and dominates the baseline afterward, showing the effectiveness of PNMAX's DSE process.

C. Architecture Exploration

We next use PNMAX to systematically explore architectural design choices jointly with mapping decisions. We explore various PNM architecture knobs with PPA-guided mapping optimization to show the comprehensive PPA impact of PNM architecture, accounting for mapping optimization. This exploration was previously difficult to do systematically, as different PNM architectures typically use different fixed-cost models and evaluation flows. In contrast, PNMAX lets users compare the interaction between architectural decisions and mapping strategies by simply supplying different GPAMs and reusing the same exploration framework. More broadly, the

experiments demonstrate the ability of PNMAX to perform architecture feature studies: a designer can describe any architecture or architectural feature in GPAM, run the PNMAX flow, and evaluate the energy, latency, and area effects. Such systematic studies and the resulting findings may impact future PNM designs.

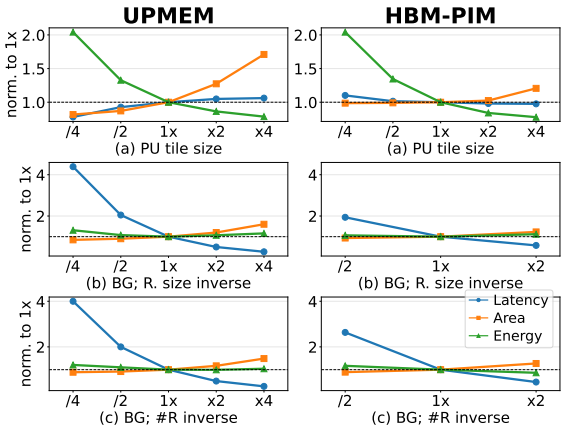


Fig. 14. Fixed-capacity PNM-geometry sweeps for OPT-2.7B on UPMEM and HBM-PIM. Each row varies one knob while fixing total DRAM capacity: (a) PU tile size (burst length + concurrency); (b) bank-group size (BG) with row size (R. size) inverse; (c) bank-group size with number of rows (#R) inverse. Latency, area, and energy are each normalized to the baseline architecture using best-latency mappings.

1) DRAM Geometry Exploration: We instantiate multiple geometry variants for both UPMEM and HBM-PIM while keeping total DRAM capacity fixed, and feed the resulting GPAMs into PNMAX to evaluate benchmark OPT-2.7B with latency as the optimization objective. We vary four primary geometric parameters: number of rows in a bank (#R), row size (R. size), bank group size (BG size), and the burst length (BL). Figure 14 presents the energy, latency, and area for these variants, each normalized to the baseline architecture (Table III) using the best-latency mapping found by PNMAX. Since total DRAM capacity is fixed, each bank-group size sweep is paired with an inverse knob: in (b) the bank row size shrinks as BG grows, and in (c) the number of rows of a bank shrinks as BG grows. HBM-PIM is shown over a narrower BG-size range, as its extreme configurations are not modeled in DreamRAM [49]. Note that increasing bank-group size increases the number of PUs simultaneously, as each bank (or each two banks for HBM-PIM) pairs with one PU. Also, the PU tile size corresponds to the burst size, so changing the burst length also changes the PU tile size accordingly.

Our experiments show a consistent trend: **increasing bank-group size (BG) while reducing the number of rows (#R) in a single bank yields the most favorable overall trade-off on both architectures**, as shown in Figure 14 (c). The more parallelism provided by more PUs and smaller operation latencies for smaller banks result in the speedup. However, increasing PU tile size (Figure 14 (a)) also increases burst length, resulting in longer DRAM operation latency and thus

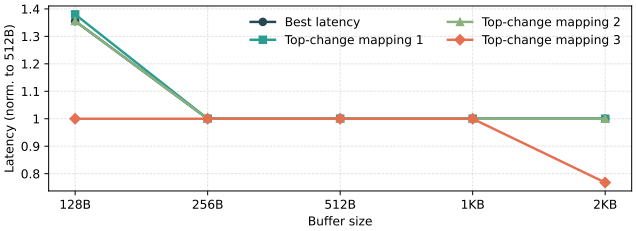


Fig. 15. Sensitivity study of buffer size for various mappings of BERT layer 72 on HBM-PIM.

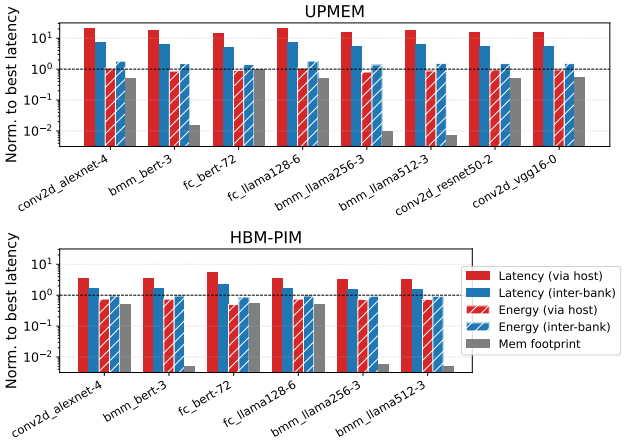


Fig. 16. Latency, energy, and memory footprint of bank-to-bank transfer via host vs. extra inter-bank interconnect, normalized to best latency mappings.

does not improve latency. The benefit larger PU tile sizes bring is less energy spent, as the number of DRAM operations decreases due to larger PU tile size. **Insight: purely increasing the number of MACs in PNM does not guarantee latency improvement due to memory costs.**

2) Buffer Size Exploration: We use PNMAX to study how buffer size affects latency across different mapping strategies, including the best-latency mappings and 3 most sensitive mappings to buffer size difference, by sweeping the buffer size attribute in the GPAM and rerunning mapping DSE for each configuration. Figure 15 shows the latency change relative to the 512B baseline buffer on HBM-PIM. Larger buffers generally reduce latency, but the sensitivity varies significantly across mapping strategies. The energy follows a similar trend to latency, as buffer hits reduce both latency and energy cost. Also, the extra buffer (2KB) adds $< 1.51\%$ area overhead compared with the DRAM stack.

3) Interconnect Exploration: PNMAX can be used to study the impact of custom interconnects by modifying connection patterns in the GPAM without changing the evaluation flow. For interconnects, we evaluate two types of inter-PU data transfer, which is necessary to enable system-level sharing: (1) host, where the host reads from one bank and writes to another bank for bank to bank transfer, without changing the architecture; and (2) inter-bank, where banks across the whole

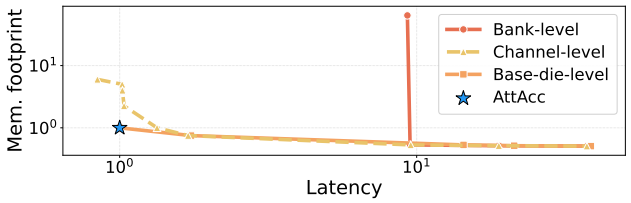


Fig. 17. Latency and memory footprint Pareto fronts for bank-, channel-, and base-die-level reduction for llama256-3 on HBM-PIM, normalized to AttAcc.

PNM system can transfer data with a direct interconnect. We conduct the experiments using all workloads on UPMEM and HBM-PIM in Figure 16, showing the best latency mapping affected by enabling system-level sharing, and the resulting latency, energy, and memory footprint changes. The results show that enabling system-level sharing reduces memory footprint by 67%, but induces 10.5× latency overhead on average via host. Meanwhile, extra inter-bank interconnect only induces 3.2× latency overhead for system-level sharing with a 53% energy and <1% area overhead compared with inter-PU data transfer via host due to the long interconnect required to connect all banks. Larger workloads (e.g., batched matrix multiplication) tend to be affected more by system-level sharing, as they need more memory capacity, and thus result in higher inter-bank transfer rates for system-level sharing. **Insight: adding extra interconnect can effectively reduce latency overhead of mappings with system-level sharing that reduces memory footprint.**

4) Reduction Placement Exploration: GPAM also supports different PU placements, and can be configured to compare different places to perform reduction. We conduct experiments in Figure 17 to compare three different reduction placements: 1) the original PU tied with the banks, 2) additional channel-level PU, and 3) additional base die PU. Each reduction level restricts the output partial sums to be partitioned below that level, ensuring partial-sum locality. Even though lower-level reduction increases the reduction parallelism, it has a more restricted mapping space than higher-level reduction. The results show that channel-level reduction can achieve the best latency due to more parallelism compared with base-die level reduction (AttAcc result) and thus less output transmission cost. However, bank-level reduction provides inferior latency due to restricted mapping space, despite providing more parallelism than the other reduction placements. **Insight: Neither the most flexible (i.e., base-die) nor the most parallel design (i.e., bank) results in the best latency. Designers need to perform mapping-architecture DSE to find the sweet spot between mapping flexibility and parallelism.**

D. Extending PNMAX to Multi-Layer Optimization

This work focuses on the mapping-architecture co-optimization of single layers on PNM architectures. Yet, we note that PNMAX can serve as the basis of multi-layer optimization. In the settings of our baselines [7], [10], all layers execute sequentially, and each of them can access all available PNM resources. Thus, decisions on different layers do not affect each other because the host reduces the final output of each layer and redistributes the data for the next layer, modeled by reduction and input loading costs in our model. In another setting where all layers share the memory space, OptiPIM [40] proposes a dynamic programming multi-layer optimization that optimizes latency for each layer given per-layer memory footprint constraints. In this case, PNMAX can optimize the per-layer latency under memory footprint constraints using the optimization method in Section IX-B, filtering out mappings with excessive memory footprint. The per-layer costs then compose the multi-layer costs. Other methods [77], [78] can also solve the multi-layer optimization problem for PIM, but are orthogonal to our contribution.

X. Related Work

Software-Hardware Co-Design of Digital PIM. Digital PIM accelerators leverage conventional memory technologies for near-data or in-situ computation [7], [9], [10], [12]–[14], [19], [20], [39], [44], [79]–[82], but rely on manual mapping and architecture tuning. Automated frameworks [7], [40], [76] remain limited: UniNDP [7] models limited digital-PIM structures and depends on a fine-grained addressing scheme that restricts scalable, analytical exploration, while OptiPIM [40] and Cinnamon [76] optimize software mapping alone. PNMAX overcomes these gaps with efficient co-exploration of both mapping and architecture for general digital PIM systems.

Software-Hardware Co-Design of Analog PIM. Analog PIM DSE frameworks [42], [83]–[87] exploit analog behaviors and assume tile-based architectures. CiMLoop [42] maps loops across analog compute and interconnects; PIMapping [85] optimizes dataflow and inter-tile communication. Yet, they cannot capture digital-PIM architectures with conventional memory hierarchies.

Software-Hardware Co-Design of ASIC Accelerators. ASIC co-design frameworks [52]–[55], [58], [88], [89] target spatial architectures with multi-level buffers and rely on exhaustive or numerical search. Their abstractions cannot capture PIM-specific behavior, such as detailed memory layouts and hierarchy-driven execution.

XI. Conclusion

We introduce PNMAX, a mapping-architecture co-exploration framework for jointly exploring mappings and architectures in processing-near-memory (PNM) systems. It provides a parameterized architecture representation spanning the PNM design space, a mapping representation that explicitly captures tensor data layout and operation scheduling, and a PPA engine for accurate cost evaluation. We show that PNMAX finds superior tensor workload mappings compared to prior frameworks and enables systematic architectural exploration under the accurate PPA constraints. Its extensible design makes it a practical foundation for developing next-generation PNM architectures.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," arXiv:1810.04805, 2019.

[3] A. Grattafiori *et al.*, "The Llama 3 herd of models," arXiv:2407.21783, 2024.

[4] OpenAI, "GPT-4 technical report," arXiv:2303.08774, 2024.

[5] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "AI and memory wall," *IEEE Micro*, vol. 44, no. 3, pp. 33–39, 2024.

[6] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference (ISSCC)*, 2014, pp. 10–14.

[7] T. Xie, Z. Zhu, B. Li, Y. He, C. Li, G. Sun *et al.*, "UniNDP: A unified compilation and simulation tool for near DRAM processing architectures," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025, pp. 624–640.

[8] S.-S. Park, K. Kim, J. So, J. Jung, J. Lee, K. Woo *et al.*, "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.

[9] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo *et al.*, "Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 43–56.

[10] J. Park, J. Choi, K. Kyung, M. J. Kim, Y. Kwon, N. S. Kim *et al.*, "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)*, 2024, pp. 103–119.

[11] O. Mutlu, "Memory-centric computing," arXiv:2305.20000, 2023.

[12] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. a. D. Ferreira, N. M. Ghiasi, M. Patel *et al.*, "SIMDRAM: A framework for bit-serial SIMD processing using DRAM," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 329–345.

[13] G. F. Oliveira, A. Olgun, A. G. Yağlıkçı, F. N. Bostancı, J. Gómez-Luna, S. Ghose *et al.*, "MIMDRAM: An end-to-end processing-using-DRAM system for high-throughput, energy-efficient and programmer-transparent multiple-instruction multiple-data computing," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 186–203.

[14] S. Rhyner, H. Luo, J. Gómez-Luna, M. Sadrosadati, J. Jiang, A. Olgun *et al.*, "PIM-Opt: Demystifying distributed optimization algorithms on a real-world processing-in-memory system," in *2024 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2024, pp. 201–218.

[15] Y. He, H. Mao, C. Giannoula, M. Sadrosadati, J. Gómez-Luna, H. Li *et al.*, "PAPI: Exploiting dynamic parallelism in large language model decoding with a processing-in-memory-enabled computing system," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025, pp. 766–782.

[16] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park *et al.*, "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.

[17] Y. Kwon, K. Vladimir, N. Kim, W. Shin, J. Won, M. Lee *et al.*, "System architecture and software stack for GDDR6-AiM," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, 2022, pp. 1–25.

[18] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son *et al.*, "A 20nm 6GB function-in-memory DRAM, based on HBM2 with a 1.2TFLOPS programmable computing unit using bank-level parallelism, for machine learning applications," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 350–352.

[19] M. Zhou, W. Xu, J. Kang, and T. Rosing, "TransPIM: A memory-based acceleration via software-hardware co-design for transformer," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 1071–1085.

[20] Y. Gu, A. Khadem, S. Umesh, N. Liang, X. Servot, O. Mutlu *et al.*, "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)*, 2025, pp. 862–881.

[21] G. Heo, S. Lee, J. Cho, H. Choi, S. Lee, H. Ham *et al.*, "NeuPIMs: NPU-PIM heterogeneous acceleration for batched LLM inferencing," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024, pp. 722–737.

[22] C. Li, Z. Zhou, S. Zheng, J. Zhang, Y. Liang, and G. Sun, "SpecPIM: Accelerating speculative inference on PIM-enabled system via architecture-dataflow co-exploration," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024, pp. 950–965.

[23] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer *et al.*, "Application-transparent near-memory processing architecture with memory channel network," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 802–814.

[24] B. Asgari, R. Hadidi, J. Cao, D. E. Shim, S.-K. Lim, and H. Kim, "Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 908–920.

[25] J. Chen, J. Gómez-Luna, I. El Hajj, Y. Guo, and O. Mutlu, "SimplePIM: A software framework for productive and efficient processing-in-memory," in *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2023, pp. 99–111.

[26] G. Dai, Z. Zhu, T. Fu, C. Wei, B. Wang, X. Li *et al.*, "DIMMining: Pruning-efficient and parallel graph mining on near-memory-computing," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022, pp. 130–145.

[27] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 283–295.

[28] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 113–124.

[29] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and efficient neural network acceleration with 3D memory," in *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 751–764.

[30] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril *et al.*, "RecNMP: Accelerating personalized recommendation with near-memory processing," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 790–803.

[31] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 740–753.

[32] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka *et al.*, "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.

[33] M. Seo, X. T. Nguyen, S. J. Hwang, Y. Kwon, G. Kim, C. Park *et al.*, "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)*, 2024, pp. 545–560.

[34] W. Sun, Z. Li, S. Yin, S. Wei, and L. Liu, "ABC-DIMM: Alleviating the bottleneck of communication in DIMM-based near-memory processing with inter-DIMM broadcast," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 237–250.

[35] B. Tian, Q. Chen, and M. Gao, "ABNDP: Co-optimizing data access and load balance in near-data processing," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023, pp. 3–17.

[36] B. Tian, Y. Li, L. Jiang, S. Cai, and M. Gao, "NDPBridge: Enabling cross-bank coordination in near-DRAM-bank processing architectures,"

in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 628–643.

[37] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang *et al.*, "SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 570–583.

[38] Y. Kwon, G. Kim, N. Kim, W. Shin, J. Won, H. Joo *et al.*, "Memory-centric computing with SK hynix's domain-specific memory," in *2023 IEEE Hot Chips 35 Symposium (HCS)*, 2023, pp. 1–26.

[39] H. Lee, D. Baek, J. Son, J. Choi, K. Moon, and M. Jang, "PAISE: PIM-accelerated inference scheduling engine for transformer-based LLM," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025, pp. 1707–1719.

[40] J. Liu, M. Zhou, Y. Pan, C.-Y. Yang, L. Josipović, and T. Rosing, "OptiPIM: Optimizing processing-in-memory acceleration using integer linear programming," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, 2025, pp. 867–883.

[41] M. Zhou, G. Chen, M. Imani, S. Gupta, W. Zhang, and T. Rosing, "PIM-DL: Boosting DNN inference on digital processing in-memory architectures via data layout optimizations," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021.

[42] T. Andrulis, J. S. Emer, and V. Sze, "CiMLoop: A flexible, accurate, and fast compute-in-memory modeling tool," in *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2024, pp. 10–23.

[43] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *10th International Workshop on Frontiers in Handwriting Recognition (IWFHR)*, 2006.

[44] F. Devaux, "The true processing in memory accelerator," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019, pp. 1–24.

[45] B. Hyun, T. Kim, D. Lee, and M. Rhu, "Pathfinding future PIM architectures by demystifying a commercial PIM technology," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 263–279.

[46] C.-S. Oh, K. C. Chun, Y.-Y. Byun, Y.-K. Kim, S.-Y. Kim, Y. Ryu *et al.*, "A 1.1V 16GB 640GB/s HBM2E DRAM with a data-bus window-extension technique and a synergetic on-die ECC scheme," in *2020 IEEE International Solid-State Circuits Conference (ISSCC)*, 2020, pp. 330–332.

[47] B. Keeth, R. J. Baker, B. Johnson, and F. Lin, *DRAM Circuit Design: Fundamental and High-Speed Topics*. Hoboken, NJ, USA: John Wiley & Sons, 2007.

[48] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 33–38.

[49] V. Cai, J. Zhou, H. Do, D. Brooks, and G.-Y. Wei, "DreamRAM: A fine-grained configurable design space modeling tool for custom 3D die-stacked DRAM," in *2026 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2026, to appear. [Online]. Available: https://arxiv.org/abs/2512.12106

[50] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "ComputeDRAM: In-memory compute using off-the-shelf DRAMs," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 100–113.

[51] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko *et al.*, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 185–197.

[52] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara *et al.*, "Timeloop: A systematic approach to DNN accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 304–315.

[53] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak *et al.*, "Interstellar: Using Halide's scheduling language to analyze DNN accelerators," in *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 369–383.

[54] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel *et al.*, "CoSA: Scheduling by constrained optimization for spatial accelerators," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 554–566.

[55] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, "ZigZag: Enlarging joint architecture-mapping design space exploration for DNN accelerators," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1160–1174, 2021.

[56] N. Nayak, X. Wu, T. O. Odemuyiwa, M. Pellauer, J. S. Emer, and C. W. Fletcher, "FuseMax: Leveraging extended einsums to optimize attention accelerator design," in *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 1458–1473.

[57] M. Gilbert, Y. N. Wu, J. S. Emer, and V. Sze, "LoopTree: Exploring the fused-layer dataflow accelerator design space," *IEEE Transactions on Circuits and Systems for Artificial Intelligence*, vol. 1, no. 1, pp. 97–111, 2024.

[58] C. Hong, Q. Huang, G. Dinh, M. Subedar, and Y. S. Shao, "DOSA: Differentiable model-based one-loop search for DNN accelerators," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023, pp. 209–224.

[59] Q. Huang, P.-A. Tsai, J. S. Emer, and A. Parashar, "Mind the gap: Attainable data movement and operational intensity bounds for tensor algorithms," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 150–166.

[60] L.-N. Pouchet, *PolyBench: The Polyhedral Benchmark Suite*, [Online]. Available: https://web.cs.ucla.edu/~pouchet/software/polybench/, 2012.

[61] "Pnmax," [Online]. Available: https://github.com/doctryucsd/PNMAX, 2026.

[62] 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 Transactions on Architecture and Code Optimization*, vol. 14, no. 2, pp. 14:1–14:25, 2017.

[63] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, "FPnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4, pp. 774–787, 2021.

[64] Google, "XLS: Accelerated hardware synthesis," [Online]. Available: https://github.com/google/xls, 2020.

[65] Cadence Design Systems, "Genus synthesis solution," [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html, 2025.

[66] ——, "Innovus implementation system," [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html, 2025.

[67] North Carolina State University, "NanGate 45nm open cell library," [Online]. Available: https://eda.ncsu.edu/freepdk/freepdk45/, 2011.

[68] S. Sarangi and B. Baas, "DeepScaleTool: A tool for the accurate estimation of technology scaling in the deep-submicron era," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[69] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: A compiler framework for analyzing and tuning memory behavior," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 4, pp. 703–746, 1999.

[70] JEDEC Solid State Technology Association, "High bandwidth memory (HBM) DRAM," JEDEC, Tech. Rep. JESD235B, 2013. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd235b

[71] ——, "DDR4 SDRAM," JEDEC, Tech. Rep. JESD79-4, 2012. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd79-4c

[72] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv:1409.1556, 2015.

[73] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[74] Hugging Face, "GPT-2," [Online]. Available: https://huggingface.co/openai-community/gpt2, 2024.

[75] ——, "OPT-2.7B," [Online]. Available: https://huggingface.co/facebook/opt-2.7b, 2022.

[76] A. A. Khan, H. Farzaneh, K. F. A. Friebel, C. Fournier, L. Chelini, and J. Castrillon, "CINM (cinnamon): A compilation infrastructure for heterogeneous compute in-memory and compute near-memory paradigms," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024, pp. 31–46.

[77] X. Sun, X. Wang, W. Li, Y. Han, and X. Chen, "PIMCOMP: An end-to-end DNN compiler for processing-in-memory accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 5, pp. 1745–1759, 2025.

[78] J. Wang, H. Du, B. Ding, Q. Xu, S. Chen, and Y. Kang, "DDAM: Data distribution-aware mapping of CNNs on processing-in-memory systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 28, no. 3, pp. 36:1–36:30, 2023.

[79] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "FloatPIM: In-memory acceleration of deep neural network training with high precision," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 802–815.

[80] Y. Pan, Z. Xia, P.-K. Hsu, L. Hu, H. Kim, J. Sharda, M. Zhou, N. S. Kim, S. Yu, T. Rosing *et al.*, "Stratum: System-hardware co-design with tiered monolithic 3d-stackable dram for efficient moe serving," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, 2025, pp. 1–17.

[81] Y. Pan, M. Zhou, C. Lee, Z. Li, R. Kushwah, V. Narayanan, and T. Rosing, "Primate: Processing in memory acceleration for dynamic token-pruning transformers," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 557–563.

[82] E. Kwon, M. Zhou, W. Xu, T. Rosing, and S. Kang, "Rl-ptq: Rl-based mixed precision quantization for hybrid vision transformers," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[83] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams *et al.*, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 715–731.

[84] R. Pelke, J. Cubero-Cascante, N. Bosbach, F. Staudigl, R. Leupers, and J. M. Joseph, "CLSA-CIM: A cross-layer scheduling approach for computing-in-memory architectures," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.

[85] Z. Zhu, Y. Zhou, J. Zhu, Y. Wang, and H. Pan, "PIMapping: A tile-level dataflow optimization framework for PIM architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2025.

[86] S. Qu, S. Zhao, B. Li, Y. He, X. Cai, L. Zhang *et al.*, "CIM-MLC: A multi-level compilation stack for computing-in-memory accelerators," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024, pp. 185–200.

[87] Z. Zhu, H. Sun, K. Qiu, L. Xia, G. Krishnan, G. Dai *et al.*, "MNSIM 2.0: A behavior-level modeling tool for memristor-based neuromorphic computing systems," in *Proceedings of the 2020 Great Lakes Symposium on VLSI (GLSVLSI)*, 2020, pp. 83–88.

[88] J. Cai, Z. Wu, S. Peng, Y. Wei, Z. Tan, G. Shi *et al.*, "Gemini: Mapping and architecture co-exploration for large-scale DNN chiplet accelerators," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 156–171.

[89] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "MAESTRO: A data-centric approach to understand reuse, performance, and hardware cost of DNN mappings," *IEEE Micro*, vol. 40, no. 3, pp. 20–29, 2020.

APPENDIX

A. Abstract

This appendix walks through installing and running the PNMAX artifact to reproduce its key experimental results: validation (Figure 8 and Table IV) against UniNDP [7] and AttAcc [10], mapping design space exploration (DSE) together with its layout and sharing analyses (Figure 10, 11, 12), PNM geometry exploration (Figure 14), buffer size sensitivity DSE (Figure 15), interconnect analysis (Figure 16), reduction units placement exploration (Figure 17), and the AttAcc [10] area analysis (Section IX-A). The artifact provides the integration of PNMAX's GPAM representation, mapping DSE infrastructure, the analytical cost model, and the per-component PPA characterization database using CACTI-7 [62], CACTI-3DD [48], DreamRAM [49], and commercial EDA tool results implementing FPnew [63] using Nangate45 [67] with Genus v25.10 [65] performing synthesis and Innovus v25.10 [66] performing place and route. The artifact includes pre-generated results; therefore, there is no need to use commercial tools to reproduce them.

B. Artifact check-list (meta-information)

- **Program:** C++, Python 3.10
- **Compilation:** g++/gcc $\geq$ 11 and CMake; clang, ninja, and git for the CINM baseline (built by default)
- **Software:** `uv`-managed Python environment; no GPU
- **Hardware:** x86-64 Linux; a many-core machine is strongly recommended (full-scale runs pin 64 workers; the reference machine has 64 cores/128 threads and 768 GB RAM).
- **Model:** AlexNet [1], ResNet [73], VGG [72], BERT [2], OPT-2.7B [75], GPT-2 [74], and Llama-3-8B [3]
- **Output:** CSV/JSON values and PDF files corresponding to Figure 8, 10, 11, 12, 14, 15, 16, 17, Table IV, and AttAcc area analysis (Section IX-A)
- **Experiments:** PPA characterization, mapping DSE, and simulator validation
- **How much disk space required?** $\approx$30 GB
- **How much time is needed to prepare workflow?** a few minutes.
- **How much time is needed to complete experiments?** $\approx$15 h at 64 workers on AMD Ryzen Threadripper PRO 9985WX 64-Cores.
- **Publicly available?** Available on
  - Zenodo: https://doi.org/10.5281/zenodo.21498745
- **Code licenses (if publicly available)?** MIT License
- **Workflow automation framework?** a top-level `Makefile`: `make setup`, then `make figxxx` for each result.

C. Description

This artifact contains the source code of PNMAX's mapping-DSE framework, links to baseline simulators, and the automation scripts to run the experiments for every result listed in Appendix B. A top-level `Makefile` drives the whole flow: `make setup` builds the environment and generates the PPA characterization database, and one `make` target per result (e.g. `make fig10`) then generates the corresponding figure or value.

1) How to access: We have archived the code on Zenodo, which supports downloading the ZIP file of this artifact. The `unzip` command is required after downloading the `PNMAX.zip` file:

```
$ unzip PNMAX.zip
```

`README.md` documents the repository layout.

2) Software dependencies: Building the artifact needs a C/C++ toolchain, CMake, and Python 3.10. `uv` manages the Python environment from the locked `uv.lock`; `make setup` bootstraps it if missing. `make setup` also builds the CINM [76] baseline, which additionally needs git, ninja, and clang/clang++.

3) Models: The nine tensor kernels (Conv2D, FC, batched-FC, MHA) are drawn from AlexNet [1], ResNet [73], VGG [72], BERT [2], and Llama-3-8B [3]; the end-to-end workloads are OPT-2.7B [75], GPT-2 [74], and Llama-3-8B [3]. All workloads are in the repository as YAML files under `data/workloads/`.

D. Installation

Before building and running the scripts, make sure the following are in place (the `README.md` has fuller instructions):

1) Library Dependencies: On most Linux distributions these dependencies come as standard packages. PNMAX depends on a working C/C++ compiler and linker, CMake to build the vendored simulators, Python 3.10, and common command-line tools such as git. The CINM baseline build additionally needs `clang` and `ninja`.

```
$ sudo apt-get update
$ sudo apt-get install -y build-essential cmake
  curl git gzip unzip clang ninja-build
```

2) Python: PNMAX manages its Python dependencies with `uv`; no manual virtual environment is needed. `make setup` bootstraps `uv` if it is missing and installs the locked dependencies with `uv sync`. Ensure Python 3.10 is available.

3) Setup PNMAX: A single command performs the one-time bootstrap:

```
$ make setup
```

This command conducts the following steps: ① Bootstrap `uv` and create the Python 3.10 environment (`uv sync`). ② Verify the three baseline GPAM specs (`hbm_pim`, `upmem`, `attacc`). ③ Build the AttAcc/Ramulator2 simulator. ④ Build CACTI and run the CACTI/DreamRAM characterization to generate the PPA database and the derived sweep architectures. ⑤ Build the CINM baseline, which needs `ninja` and `clang`; this LLVM build is a long step.

E. Sample Test of PNMAX

To verify the installation, run the smoke suite, which exercises every pipeline end-to-end in about 12 minutes:

```
$ make smoke
```

Each script prints a final `<experiment> done` banner on success, and `attacc_area` additionally prints `CHECK OK`.

F. Experiment workflow

1) Notes: The full campaign takes ≈15 h at the default 64 workers on the reference machine, a 64-core/128-thread AMD Ryzen Threadripper PRO 9985WX with 768 GB RAM. Figures 10 and 14 dominate at ≈6.5 h each; Figure 17 takes ≈1.6 h, Figure 8 ≈5 min, and the Section IX-A check seconds. Outputs are under `results/<experiment>/`, and configurable through setting `PNMAX_RESULTS_ROOT`. To handle accidents during the long runs, all the interrupted scripts are safe to resume and the runs will continue on the previous partial results.

Reproduced numbers and figures might differ slightly from those in the paper because the paper's DSE runs were unseeded. The Pareto front shapes and values might vary. However, the overall trends, and thus the paper's conclusions, should still hold.

2) Figure 8 and Table IV: To generate Figure 8 and Table IV, run:

```
$ make fig8
```

This command will: ① Generate mapping sets from the evaluation kernels. ② Run the mappings through the UniNDP [7] and AttAcc [10] simulators and the PNMAX model. ③ Produce `fig08.pdf` with their R^2 on top, and the Table IV wall-time and correlation values.

3) Figure 10 and its mapping search-related analyses: Run the Figure 10 target first; it builds the shared mapping search pool that the four pool-derived targets reuse:

```
$ make fig10   # run first
$ make fig11
$ make fig12
$ make fig15
$ make fig16
```

The first command will: ① Derive the 18 UniNDP baseline mappings. ② Run the mapping DSE for the nine kernels on UPMEM [44] and HBM-PIM [9] over pragma spaces (a)–(d). ③ Plot the Pareto fronts against the UniNDP [7], OptiPIM [40], and CINM [76] baselines as `fig10.pdf`. The streaming (Figure 11), latency-breakdown (Figure 12), buffer-sensitivity (Figure 15), and bank-to-bank interconnect (Figure 16) scripts reuse the mapping pool.

4) Figures 14 and 17: To generate the two architecture-exploration figures, run:

```
$ make fig14
$ make fig17
```

Figure 14 sweeps the PU tile size, burst length, and bank-group geometry for OPT-2.7B on UPMEM and HBM-PIM. Figure 17 explores bank-, channel-, and base-die-level reduction placement for `llama256-3` on HBM-PIM.

5) AttAcc area analysis (Section IX-A): To reproduce the Section IX-A area value, run:

```
$ make attacc_area
```

This command computes the AttAcc PU+buffer area overhead from the PPA database, and prints the derivation and the value.

G. Evaluation and expected results

All evaluation results, the CSV/JSON values and the PDF figures, are located under `results/<experiment>/` as specified in Appendix F; the `README.md` lists the result files of each experiment. The expected results are the figures and trends reported in the paper. A successful reproduction shows:

- **Figure 8 + Table IV:** every scatter subplot reports a high R^2 (near 1), and the model runs faster than every simulator; the speedups are machine-dependent, so only the direction reproduces.

- **Figure 10:** the fronts improve as pragma spaces (a)–(d) are added and dominate the UniNDP baseline ($\times$), OptiPIM (star) and CINM (diamond) in all 18 subplots.
- **Figure 11:** disabling streaming increases the memory footprint of every mapping on both architectures; UPMEM's latency delta is much smaller than HBM-PIM's.
- **Figure 12:** improvements from (a) to (b) come from reduced output cost; improvements from (c) to (d) come from either reduced input streaming or reduced bank-to-pu load.
- **Figure 14:** The generated latency, energy, and area trends should look like those in the paper.
- **Figure 15:** The sensitivity of the mappings to buffer size changes should differ.
- **Figure 16:** bank-to-bank interconnect via host induces more latency but less energy than extra bank-to-bank interconnect with the same memory footprint reduction. The number of workloads shown might be different since the generated figure shows only workloads with memory footprint improvement.
- **Figure 17:** channel-level reduction achieves the best latency, base-die ($\approx$ AttAcc) is next, and bank-level is worst.
- **Section IX-A:** `computed_overhead_pct` $\approx$ 7.85% (vs. 10.84% in the AttAcc literature).