PIM-Potential: Broadening the Acceleration Reach of PIM Architectures

Johnathan Alsop
johnathan.alsop@amd.com
AMD Research

Shaizeen Aga
shaizeen.aga@amd.com
AMD Research

Mohamed Ibrahim
mohamed1.ibrahim@amd.com
AMD Research

Mahzabeen Islam
mahzabeen.islam@amd.com
AMD Research

Nuwan Jayasena
nuwan.jayasena@amd.com
AMD Research

Andrew Mccrabb
mccrabb@umich.edu
University of Michigan

Abstract

Continual demand for memory bandwidth has made it worthwhile for memory vendors to reassess processing in memory (PIM), which enables higher bandwidth by placing compute units in/near-memory. As such, memory vendors have recently proposed commercially viable PIM designs. While the challenge of efficient PIM orchestration requires consideration of new constraints across the compute stack, these can often be hidden from software and microarchitecture for highly regular workloads (e.g., common machine learning, or ML, primitives). However, these new constraints are not as easy to hide for workloads that exhibit certain types of irregularity. To extend PIM's reach to a broader range of workloads, navigating these new constraints becomes necessary at all levels of the compute stack.

In this work, we analyze the capabilities and constraints of a promising new type of commercial PIM architectures and we describe the properties that make a workload amenable to acceleration on such a system. Next, we explore how limitations of these PIM designs like row activation overheads, lack of reuse benefit, and command bandwidth can expose novel bottlenecks for some workloads. These workloads, termed *PIM-potential* workloads, have properties which deviate in limited ways from the identified amenability characteristics but enjoy only minor performance gain from PIM. The exposed bottlenecks motivate targeted hardware and software optimizations - eager activation, increased register storage, selective PIM command issue, and increased command bandwidth - that can be leveraged to mitigate these performance constraints and enable PIM acceleration for a wider range of workloads. We evaluate their impact on *PIM-potential* primitive acceleration and demonstrate that PIM can be applied more broadly than previously described if the emergent PIM bottlenecks can be addressed. We argue that emerging PIM architectures and programming models should take into account these novel PIM bottlenecks and corresponding optimizations in order to enhance the scope of PIM acceleration.

1 Introduction

As applications of both commercial and scientific importance continue to demand more memory bandwidth, memory vendors are reassessing processing in memory (PIM) as a potential solution. With PIM, in/near memory compute units work in tandem with traditional processors to enable higher effective memory bandwidth (potentially an order of magnitude or more) over that available externally (e.g., to CPUs, GPUs, ASICs, etc.). Recently, multiple memory vendors have proposed commercially viable near-bank PIM designs [32, 33]. Going forward, we refer to such architectures as "tightly-coupled PIM" or simply "PIM" (discussed in detail in Section 2.2 and differentiated from other types of PIM in Section 6).

The limitations of tightly-coupled PIM mean that it is only useful (and has only been evaluated) for workloads with PIM-amenable characteristics (discussed in Section 3). Rather than focusing on known-PIM-friendly workloads (i.e., extremely regular, parallel, memory bound workloads), we choose to probe the limits of PIM acceleration by selecting workloads that exhibit some but not all of the identified PIM-friendly characteristics. Via careful evaluations, we observe that while these primitives show promise in terms of PIM amenability, this *potential* is not realized in practice. We term such primitives as **PIM-Potentials**.

With further investigation, we identify unique challenges that arise in such PIM designs and prevent realization of the acceleration potential of primitives under study. First, we observe that while techniques like bank parallelism allow effective hiding of row activation overheads for a baseline system, broadcasting the same command to multiple banks exposes these overheads for a PIM system. Secondly, while a baseline system can often harness data reuse benefits with large on-chip structures (e.g., register files, caches), PIM, without large scratchpads (due to associated area overheads) fails to harness data reuse benefits. Similarly, while computation on on-chip data in a baseline system is cheap enough to tolerate unneeded compute (e.g., multiplication with zero), this is not so for PIM where every PIM command is a memory command. Finally, we observe that PIM stresses available command bandwidth in the system. We describe how these challenges can limit PIM performance for the studied primitives as well as for any other workloads that exhibit similar properties.

Finally, to address above bottlenecks, we propose targeted hardware and software optimizations and evaluate them for **PIM-Potential** primitives from different domains. Specifically, to tackle row activation overheads we propose and evaluate careful and eager scheduling of row activations in a PIM system. To tackle data-reuse disadvantage and costly-compute in PIM, we propose selective PIM command issue which offloads to PIM in a cache-aware (reuse-aware) manner and also opportunistically avoids issuing PIM commands where possible while preserving functional correctness (e.g., sparsity-aware PIM command issue). Finally, we consider the benefits of addressing command bandwidth bottleneck in PIM. We show how our proposed optimizations stand to broaden the acceleration reach of tightly-coupled PIM designs, achieving speedups of up to 2.68x, 3.17x, and 2.43x in scientific, ML, and graph analytics domains respectively (of an available upper-bound of 4x).

Overall, our work makes the following contributions:

- We introduce and focus on PIM-Potential primitives. That is, primitives which have some characteristics that make them amenable to acceleration with tightly-coupled PIM systems, however, this acceleration potential is not realized with current PIM designs.
- We identify unique constraints and bottlenecks which manifest when these primitives are offloaded to tightly-coupled PIM architectures.
- We demonstrate how simple and targeted hardware and software optimizations can unlock the acceleration potential for PIM-Potential primitives improving average PIM speedups from 1.12x to 2.49x relative to a GPU baseline.

With the above analysis, this study proves that tightly-coupled PIM can accelerate a broader range of domains than previously studied as long hardware and software are designed with these new bottlenecks in mind.

2 Background

In this section, we first discuss and motivate our assumed baseline system, our assumed PIM architecture, and the domains and primitives that we focus on in this work.

2.1 Baseline System - GPU + HBM

The left side of Figure 1 depicts the baseline system studied in this work: a GPU coupled with HBM memory [1].

GPU: While PIM can be coupled with any processor (CPUs, GPUs), our evaluation assumes a GPU for multiple reasons. First, over the past decade, GPUs have emerged as performant and programmable accelerators for a diverse range of highly parallel compute workloads. Second, there exists a real PIM prototype [33] coupled with GPUs allowing us a baseline architecture to assess. Finally, as GPU compute throughput is increasing more rapidly than memory bandwidth, many emerging GPU workloads are likely to be memory bandwidth bound.

High Bandwidth Memory (HBM): High bandwidth memory is a specialized form of DRAM that attains high bandwidth and energy efficiency via high density interconnects and 3D stacking. As illustrated in Figure 1, each HBM module is a 3D-stack of DRAM dies and a base logic die connected using low-power through silicon vias (TSVs). HBM can be tightly integrated with a processor (in this case a GPU) die on a common substrate such as a silicon interposer [31] with an order of magnitude more I/O interconnects [33] than conventional DRAM.

Each HBM DRAM die is composed of pseudo-channels (pCHs), which further comprise multiple banks that share the data bus associated with a pCH. The address associated with a baseline read/write request specifies the pCH and bank where the data resides along with the row and column address within the bank. On a read request, the specified row is activated in the bank (*row activation*), which causes the data in the row to be read out to the row-buffer associated with the bank (*row open*) after DRAM row activation delay. Subsequently, the column decoder selects a DRAM word from the row buffer based on specified column address (*column access*). Row activation delay overhead can be mitigated in DRAM by exploiting row locality (subsequent accesses to the open row do not incur activation latency) and bank parallelism (column commands to different banks keep the data bus utilized while one bank is activating a row). Note that the basic sequence of operations for an HBM memory access is similar to that of DDR DRAM.

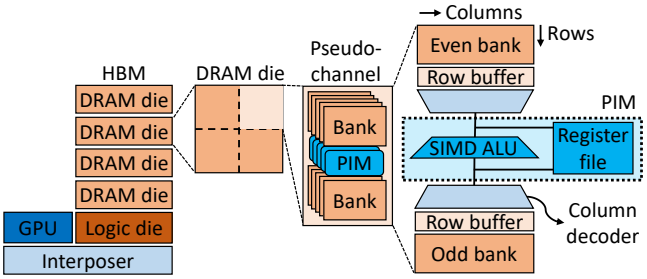

Figure 1: PIM design based on PIM-HBM [33].

2.2 Tightly-coupled PIM Architectures

Recently, two designs for DRAM-attached tightly-coupled (bank-local) PIM have been announced by Samsung and SK Hynix. We discuss the details of each design in this section.

HBM-PIM: Samsung's proposed PIM architecture [27, 33] places ALUs and associated register files on the periphery of DRAM banks (Figure 1). This design does not disturb the bank and sub-array structures of the memory, improving its viability for commercialization. The PIM ALU is a 256b-wide SIMD datapath that performs sixteen 16b operations in parallel, and is matched to the input/output width of the DRAM cell arrays of the bank. The register files can be used for intermediate results or for staging data from an open DRAM row to reduce the frequency of row activations. Notably, the PIM units do not contain instruction fetch or other "frontend" capabilities, reducing their area costs, and they execute instructions in response to commands issued from the GPU processor. These GPU commands are issued subject to fixed timing constraints, similar to how traditional memory operations are issued. The key benefit of this architecture is memory bandwidth amplification, which is achieved by broadcasting the commands to all banks (or a subset of banks) of a pCH (normal load/store operations only access a single bank at a time). This is possible because the data from each bank goes to the associated PIM unit rather than being transmitted across the shared pCH. The memory functions as a standard DRAM when the PIM capabilities are not used.

The authors also describe a prototype implementation named PIM-HBM that is fabricated as an extension to HBM2 and is evaluated in silicon. In the prototype, each PIM unit is shared between a pair of banks, demonstrating the flexibility of the design to provision a PIM unit per bank, per pair of banks, or other grouping of banks to tradeoff performance vs area/cost considerations. The PIM ALUs of the prototype support a limited but generic set of ALU operations, which can presumably be extended in subsequent implementations.

GDDR-PIM: The SK Hynix design [25, 32, 34] describes a PIM system based on GDDR6 that is specifically targeted for ML inference applications and is tailored to matrix-vector multiplications and non-linear activation functions. Despite the specific application focus, this design takes a very similar approach to the PIM-HBM in that it places compute units on the periphery of DRAM banks and relies on the GPU for instruction triggering in place of native

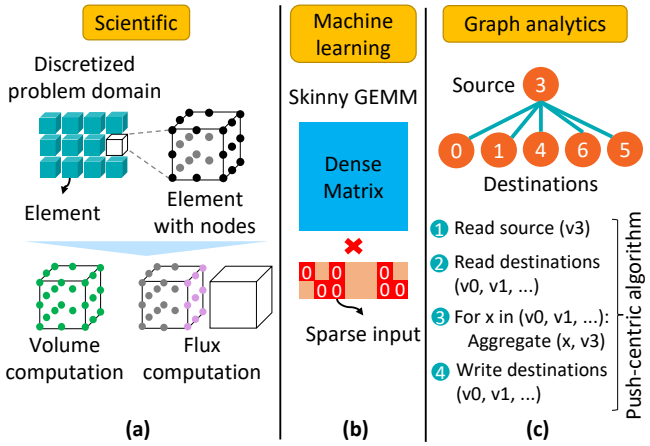

Figure 2: PIM-Potential primitives.

Table 1: Tightly-coupled PIMs relative to GPU

Property	MI250-GPU	HBM-PIM	GDDR-PIM
Mem clock (GHz)	1.6	1.2	1.0
FP16 TFLOPS	45	1.2	1
PIM data BW (GB/s)	n/a	1229	1024
Mem BW (GB/s)	400	307	64

frontend hardware. The datapath is also in a 256b SIMD configuration matched to the bank input/output data width. In the evaluated prototype implementation, a PIM unit is instantiated for each bank of the GDDR6 memory module.

Tightly-coupled PIM Performance Space: Table 1 provides key performance metrics for the above tightly-coupled PIM designs (per-device compute and data bandwidth). We also include state-of-the-art GPU (AMD Instinct™ MI250 Accelerator) performance metrics (per-HBM stack) for comparison. As depicted, PIM data bandwidth is considerably higher than memory bandwidth available to the GPU, while GPU compute capability is considerably higher than that of PIM.

PIM Strawman: For our analyses, we distill a near-bank high-bandwidth PIM design based on the basic characteristics common to the two memory vendor PIM proposals above but lean closer to HBM-PIM for two reasons. First, HBM-PIM is the more flexible of the two in terms of programmability, and our interest is in further broadening the applicability of PIM. Second, as many modern high-performance GPUs use HBM DRAM, HBM-PIM provides a natural comparison point. We discuss how this assumed architecture differs from previously analyzed PIM architectures in Section 6.

2.3 Domains and Primitives

We select three primitives from three important application domains: scientific computing, machine learning, and graph analytics. These primitives, which we refer to as PIM-Potential primitives, are depicted in Figure 2. They are chosen because of their importance within their respective domains, and also because they represent a type of PIM amenability that has not been thoroughly studied in the context of tightly-coupled PIM. PIM-Potential primitives have some characteristics that make them amenable to PIM (discussed in Section 3.2), but also expose new challenges that are unique to tightly-coupled PIM systems (discussed in Section 4.4).

2.3.1 Scientific - Wave Simulation. The solution of partial differential equations (PDEs) is critical to many large-scale problems in HPC systems. One such use case, wave simulation, requires solving the wave equation to model the propagation of waves through different media and is used extensively in domains including medical imaging, earthquake modeling, oil and gas exploration, and antenna and radar modeling.

The Discontinuous Galerkin Method (DGM) is a popular algorithm for wave simulation due to its scalability [51]. Like many PDE solvers, DGM discretizes the wave space into a mesh of elements which are distributed among processors in the system (Figure 2a). It then iteratively executes a volume computation and a flux computation along with communication and support computations to model wave propagation. In both *volume* and *flux*, multiple properties from neighboring data points are accumulated in multiple intermediate reduction variables. The volume computation (termed *wavesim-volume* primitive) performs computations local to each mesh element; the flux computation (termed *wavesim-flux* primitive) propagates conditions at the boundaries (faces) of each mesh element. These two computations dominate execution time for most simulation tasks and as such we focus on these in our work.

2.3.2 Machine Learning - Sparse Skinny GEMMs. Machine learning (ML) continues to become ever more pervasive. At the heart of many ML [43] workloads is General Matrix-Matrix multiplication (GEMM). Unlike prior PIM evaluations which focus on skinny GEMMs (the non-shared dimension is small) which are dense, in this work, we focus on GEMMs where one of the matrix inputs is also sparse (has many zeros, as in Figure 2b). We term these sparse skinny gemms (*ss-gemm*) and they manifest in many ML inference scenarios (e.g., Deep Learning Recommendation Model (DLRM) [41] with small batch sizes).

2.3.3 Graph Analytics - Push-based Computation. Graph analytics attempts to derive insights by analyzing the connectivity of graphs and data associated with its edges and vertices. Graph analytics is regularly used for navigation, chemical and biological modeling, social network monitoring and analysis, and many more applications.

Many common graph analytics workloads operate by iteratively propagating vertex properties (pull or push) across graph edges to neighboring vertices. In pull-based algorithms, a local vertex is processed by reading properties from each of its neighbor vertices and updating the local vertex based on what was read. In push-based algorithms (Figure 2c), a local vertex is processed by reading its properties and updating neighbor vertices based on what was read (using atomic RMWs to avoid race conditions). Push implementations have been found to offer attractive performance properties for many graph algorithms and inputs [11, 44], and they are widely used in GPU graph frameworks [21, 50] and benchmark suites [19, 39]. As such, we focus on a push-based algorithm primitive (termed *push-primitive*).

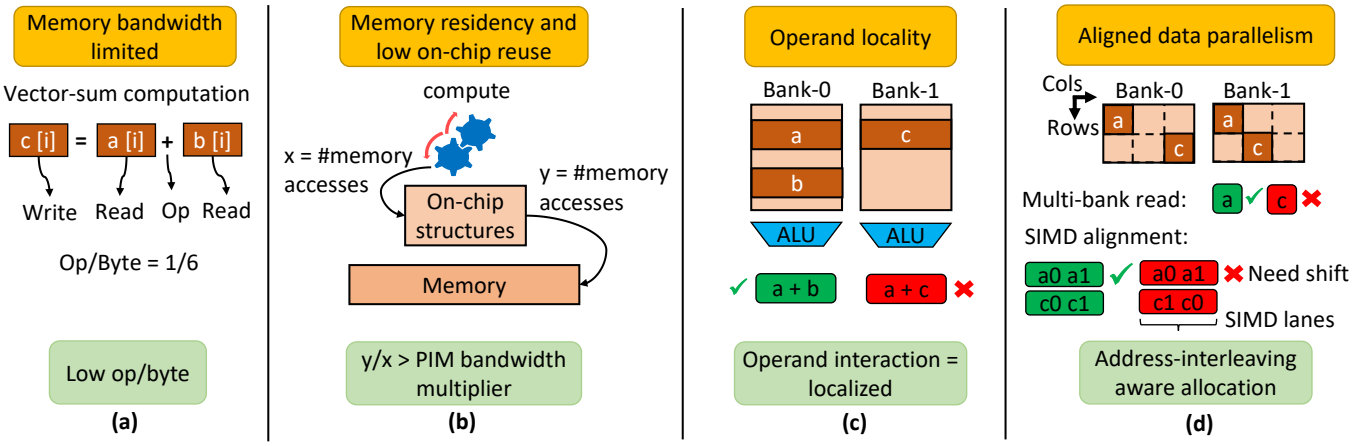

Figure 3: Characteristics of interest for PIM-amenability-test.

3 Tightly-coupled PIM-amenability

Here we describe what is necessary for a compute workload or primitive to benefit from PIM execution. We then describe the ways in which the PIM-Potential primitives exhibit PIM amenability and non-amenability.

3.1 PIM-friendly Characteristics

We begin this section by discussing a list of characteristics which help evaluate if a given computation is likely to benefit from PIM.

3.1.1 Memory bandwidth limited. PIM's primary performance benefit comes from increasing effective memory bandwidth.[1] Therefore, it will not improve performance for workloads that do not stress available memory bandwidth. Memory bandwidth sensitivity will depend on both the workload and the target architecture. This property can be tested analytically by calculating the algorithmic op/byte ratio (Figure 3a) and determining whether it falls in the compute-limited or memory-limited range on a roofline model for the target architecture.

3.1.2 Memory residency and low on-chip reuse. Even for workloads that are limited by memory bandwidth, residency and on-chip reuse of computation's inputs/outputs can preclude PIM amenability. Should a computation's inputs/outputs be resident in on-chip structures or manifest enough reuse when moved from memory to on-chip structures, the computation is less likely to attain acceleration with PIM. For the former, employing PIM would necessitate flushing data to memory, resulting in added data movement overhead. For the latter, with enough reuse, moving data closer to the processor to take advantage of the fast and low energy caches and compute available at the processor is a better solution than using slower compute available in memory. As a general rule, PIM acceleration of an access pattern is possible if the ratio of on-chip to off-chip data access is less than the PIM bandwidth multiplier.

3.1.3 Operand Locality. As discussed in Section 2.2, compute units in tightly-coupled PIM designs are associated with specific memory bank(s). As such, interacting operands in a computation should map to the same bank to effectively harness PIM acceleration (Figure 3c). We term this property *operand locality* in our discussion. In the absence of operand locality, costly GPU-orchestrated data transfers between banks will be necessary, which will eat into PIM acceleration. As an example, consider elementwise computations (e.g., vector sum) in which elements in a data structure interact with the corresponding element index in another structure. In this case, operand locality of interacting elements can be achieved via careful co-alignment of data structures at allocation [6]. For more complex structures (e.g., 2D matrices), operand locality may require other types of intelligent data mapping (e.g., padding, address swizzling). In all cases, any data mapping costs should be factored in when assessing end-to-end PIM impact.

3.1.4 Aligned Data Parallelism. The bandwidth boost attained in PIM is possible via execution of the same operation in parallel across multiple banks. As discussed in Section 2.2, as memory operations have associated row and column addresses, this bank-parallel execution can be employed when operands in different banks in a computation are at the same row/column locations (e.g., see accessing operand `a` across banks vs. accessing operand `c` in Figure 3d). Note that, within a single DRAM word (256bit single DRAM column), interacting operands therein (e.g., 32bit operands) also have to align (depicted as SIMD alignment in Figure 3d). In absence of this, costly shift operations will be necessary. We term these properties together as *aligned data parallelism*. As processors often spread a contiguous physical address chunk across multiple channels/banks, ensuring interacting PIM operands are interleaved similarly at allocation time can help attain this characteristic.

3.2 PIM-Potential Amenability

We evaluate PIM amenability of the PIM-Potential primitives (Section 2.3) by analyzing them for the characteristics described in Section 3. While these workloads exhibit many important PIM-friendly characteristics (with careful data mapping and orchestration), they also expose novel performance limitations unique to PIM (discussed in Section 4.4).

Wave Simulation: With low op/byte (0.43-1.72), wave simulation primitives under study (*wavesim-volume* and *wavesim-flux*)

[1] Some forms of PIM may offer improved memory latency as well, but this work focuses on the bandwidth benefits.

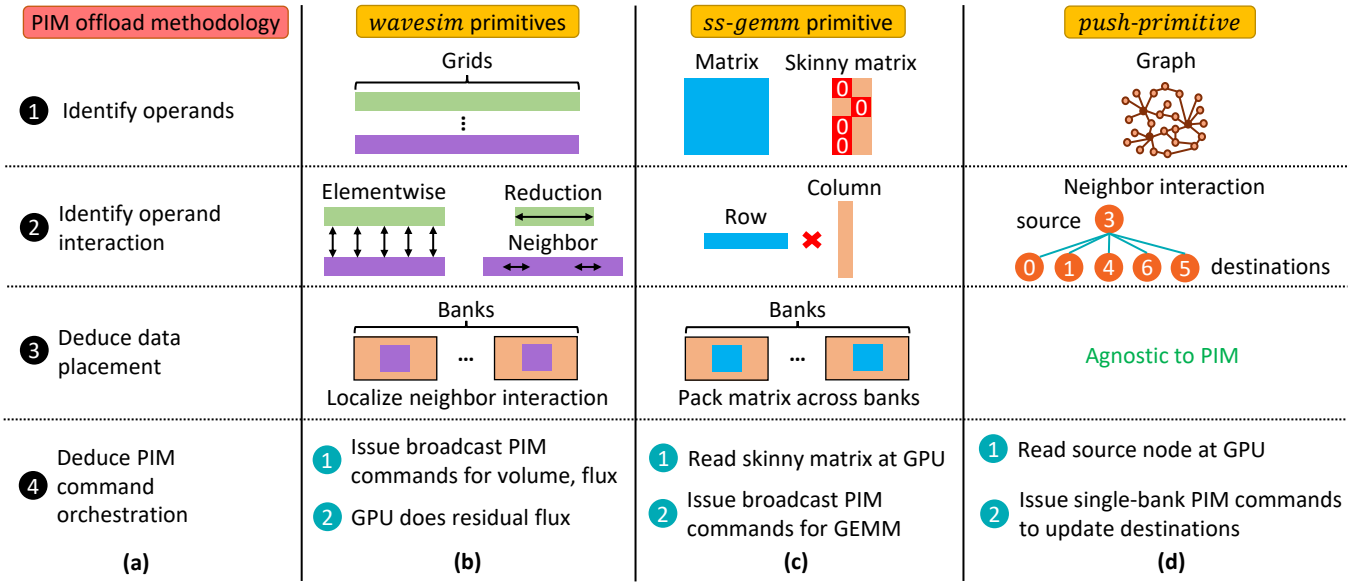

Figure 4: (a) Methodology to offload primitives to PIM. Applying proposed methodology to primitives under study (b, c, d).

are likely to be memory bandwidth-limited for most architectures. Further, for problem sizes that do not fit in cache (the common case), these primitives manifest low on-chip reuse (although some reuse exists in accumulation variables). While largely relying on localized interaction, these primitives do manifest interactions between neighboring mesh element faces (flux in Figure 2a). As such, careful memory allocation is necessary to maximize mapping of interacting neighbors to the same DRAM bank and further, not all computation can be mapped to PIM (interactions between neighboring elements in different banks). Finally, these primitives operate on large regular grids of elements which can be harnessed to achieve aligned data parallelism. Overall, wave simulation primitives show promise in terms of PIM amenability, albeit care is necessary to attain operand locality.

Sparse Skinny GEMMs: With one of the input matrices being skinny (small N for GEMM $M \times N \times K$), the *ss-gemm* primitive manifests low op/byte (0.5-2 with $N \leq 4$) and low reuse of data and can benefit from PIM for sufficiently large problem sizes (although both op/byte and reuse increase with larger N). By streaming the skinny matrix to PIM units and keeping the other matrix stationary in memory, operand locality can be simplified. Further, unless the row-size of the matrix resident in memory is considerably large, aligned data parallelism requires considerable care for *ss-gemm* and we discuss how this guided our data mapping (Section 4).

Push-based Computation: The dominant computation for *push-primitive* is the access and update of neighboring nodes (Figure 2c) which manifests a low op/byte ratio (0.25). On-chip reuse is dependent on the connectivity and size of the graph, although sufficiently large problem sizes tend to be limited by cache misses and memory bandwidth. As long as PIM is simply performing an in-place update on each target variable, operand locality is trivial. However, the irregularity of accesses to neighbor nodes precludes most aligned data parallelism. As we will discuss, this limits PIM potential for *push-primitive* (Section 4).

4 Baseline PIM

We discuss in this section how PIM-Potential primitives can be offloaded to PIM. We then evaluate PIM performance and detail novel sources of inefficiency exposed by these workloads.

4.1 Offloading Primitives to PIM

Once a programmer has determined that a primitive has PIM potential, they must determine how best to offload the operations to PIM. As discussed in Section 3.1, ensuring operand locality and aligned data parallelism is critical to attaining PIM acceleration. To that end, identifying operands or data structures (❶) and, more crucially, identifying interactions between them (❷) are the first steps in offloading to PIM. Subsequently, operands are placed in DRAM banks (❸) such that costly inter-bank communication, cross SIMD operations, and (where possible) inter-row interactions are avoided. Finally, (❹), a stream of *pim-instructions* is deduced to orchestrate the computation over PIM units.

We next describe how we place data and orchestrate PIM commands for the PIM-Potential primitives discussed previously. We also describe this for a PIM-amenable primitive (vector sum) for comparison purposes.

4.1.1 Wave Simulation. **Data placement:** Wave simulation largely operates over arrays and employs three types of operand interaction: elementwise, reduction, and neighboring mesh element. Of these, while elementwise interaction can be tackled via data placement as has been shown in past work [5], reduction and neighbor interaction warrant more care. For the former, blocked data placement is employed (discussed below in the context of *ss-gemm* primitive which also employs this). For the latter, array (grid) elements are placed such that, to the extent possible, neighboring faces reside in the same bank as depicted in Figure 4b.

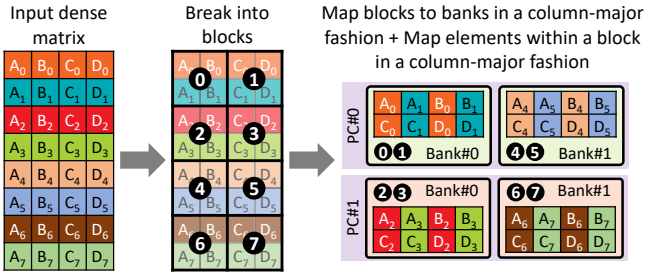

Figure 5: Workload *ss-gemm* data mapping.

Command orchestration: Despite their PIM amenable properties, *wavesim* primitives exhibit complex interaction patterns between operands which complicate orchestration. Considerable care is necessary to effectively utilize available registers while avoiding memory spills and lowering row activation overheads. While we hand schedule the computation in our analysis, existing compiler methods for register allocation [15, 17] can be adapted for PIM-specific cost awareness.

4.1.2 Sparse Skinny GEMMs. **Data placement:** To harness broadcast *pim-commands* and concomitant performance, we place the input dense matrix in a blocked format as depicted in Figure 5, which is tailored to both the dimensions of the matrix and the address interleaving of the system. This layout has multiple attractive properties: skinny matrix values can be broadcast as an immediate PIM operand on the data bus, and accumulation of partial products avoids inter-bank, intra-SIMD, and (to the extent possible) inter-row operations.

Command orchestration: Compute orchestration follows from data placement: an element of the skinny matrix is loaded into the host and broadcast to banks that contain the elements of the (larger) dense matrix. The broadcast element is then multiplied with these dense matrix elements using a vector *pim-MAC* operation at each bank. Partial results are accumulated in *pim-registers* before being written to memory.

4.1.3 Push-based Computation. **Data placement:** Variation in graph connectivity precludes the use of broadcast commands and co-location of interacting neighbors (source and destination nodes in Figure 4d). Instead, single-bank *pim-commands* execute in-place destination updates, avoiding operand locality layout constraints.

Command orchestration: Compute orchestration for *push-primitive* also follows from its data placement. The source node's value is read from memory, then updates to neighboring nodes are calculated and applied using single-bank *pim-commands* (namely, a *pim-ADD* command loads the current value and adds an operand supplied on the data bus, placing the result in a *pim-register*, and a *pim-store* command stores this result to memory).

4.2 Performance Models

We use a combination of analytical models and detailed memory timing models to evaluate performance. The use of analytical modeling is consistent with prior PIM evaluations [28, 29] and is guided by the following reasons. First, tightly-coupled PIM designs are still only available as functional prototypes. Second, we aim to study primitives considering realistic problem sizes, where PIM is likely to be beneficial. This renders GPU simulators difficult to use due to long simulation times. Finally, we target highly parallel GPU workloads that are principally bottlenecked by memory bandwidth; accurate modeling of compute and caches will not impact overall performance in such cases.

Table 2: Parameters for performance model [2].

#Banks per Channel/4-high Stack	16 / 512
Bandwidth per Pin	4.8 Gb/s
GPU Mem. Bandwidth per Stack	614.4 GB/s
Row Buffer Size	1024 B
DRAM Parameters	tRP=15 ns, tCCDL=3.33 ns, tRAS=33 ns
PIM Parameters	#PIM Units per Stack = 256 #PIM Registers per ALU = 16
Peak HBM Bandwidth	614.4 GB/s

Baseline GPU Performance Model: For our GPU baseline, we assume the execution time is primarily a function of memory bandwidth (assumed to be 90% of peak) and data accessed. As a result, compute and cache access is effectively free in our model. Further, we assume caches are able to exploit all available reuse with two exceptions: inter-timestep reuse is not modeled for *wavesim* (we assume polynomial degree $p = 2$, 729 data points per element, and $65K$ elements per GPU, which is too large to fit in cache), and cache locality for *push-primitive* is based on actual cache hit rates measured using rocprof [3] with push-based workloads from graphBIG [39] (specifically, hit rates of 44%, 20%, and 57% are observed for roadnet-usa [19], a synthetic power-law graph with 1M nodes and 10M edges, and a synthetic power-law graph with 10M nodes and 100M edges, respectively).

We believe this to be an fair assumption for memory-limited workloads which manifest low op/byte ratios as discussed in Section 3.2. Further, we assume HBM3 memory [2] in our analysis (Table 2). We do so to be both forward-looking and avail our baseline GPU with the best possible memory bandwidth. As such, this provisions a compelling baseline against which to compare PIM acceleration benefits.

Further, for *ss-gemm*, we assume an optimized GPU baseline which can exploit row-sparsity to both avoid loading the zero rows and computing on them. We estimate this sparsity by analyzing row-sparsity occurrence for computations in MLPerf DLRM-based recommendation model [41] using the Terabyte Click Logs testing dataset [18].

PIM Performance Model: For PIM, we first deduce detailed *pim-commands* (Section 4.1) and subsequently model their detailed orchestration using known DRAM timings (Table 2) and operations (row activation, etc.) to determine PIM execution time. Multi-bank *pim-commands* are issued in-order at half the rate[2] of regular reads/writes as is the case with the HBM-PIM design [33]. Single-bank *pim-commands* (used for *push-primitive*) can be freely reordered and can be issued at the same rate as regular reads/writes.

[2]As dictated by the t_{CCDL} timing parameter for back-to-back requests to the same bank group, as opposed to the minimum possible time between reads/writes: t_{CCDS}.

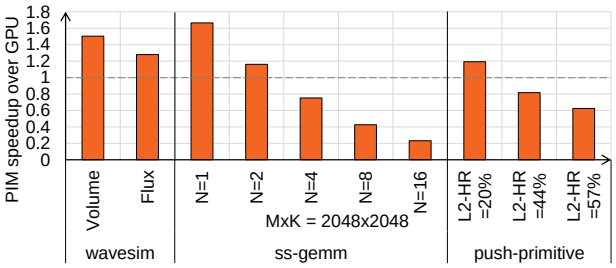

Figure 6: Tightly-coupled PIM speedup relative to GPU. For *ss-gemm*, N represents skinny matrix width. For *push-primitive*, L2-HR represents the hit rate measured at L2 cache for each graph evaluated.

Further, *push-primitive* updates are also assumed to occur atomically, which can be guaranteed by existing per-address ordering assumptions in the memory controller.

4.3 PIM Performance Analysis

We depict PIM speedups for PIM-Potential primitives in Figure 6. For our PIM strawman design, the upper bound for performance is about 4x assuming the baseline GPU can utilize 100% of peak memory bandwidth (optimistic).

For the *wavesim-volume* and *wavesim-flux* primitives, PIM offers minimal speedups, but row activation overheads prevent it from achieving peak PIM bandwidth. For the *ss-gemm* primitive, except very skinny matrices ($N = 2$), PIM incurs increasing slowdown (between 25-77%) as N increases compared to the GPU. This is expected because, as data reuse improves, moving data to the GPU and exploiting the reuse on chip is more beneficial. Although PIM enables sub-cache line access granularity which can accelerate the irregular *push-primitive* accesses when cache reuse is low, the inability to issue aligned PIM requests prevents significant PIM acceleration. In addition, as the cache hit rate improves, relative PIM performance decreases (similar to *ss-gemm*).

Overall, we observe that, even with considerable care to place data appropriately and orchestrate computation efficiently, tightly-coupled PIM designs do not attain broad acceleration for PIM-Potential workloads.

4.4 Novel Challenges to PIM Acceleration

In this section we further analyze PIM performance, focusing on unique challenges when computations are offloaded to PIM. We also discuss how the identified challenges are fundamental to tightly-coupled PIM designs and not specific to PIM-Potential primitives. That is, addressing these challenges will likely enable broad acceleration with tightly-coupled PIM.

Challenge - Row activation overhead: A key limit to PIM acceleration is row activation overheads - that is, latency costs to open DRAM rows. In a non-PIM system, row activation overheads can impact performance, but they are often mitigated via bank parallelism (row activation latency in one bank is overlapped with data access in another bank) and exploiting row locality (scheduling as many requests to an open row as possible before closing it). Note that, though pseudo-channel (pCH) level data access parallelism can be achieved by keeping all the pCHs busy at the same time (each pCH has individual data bus), to ensure higher data bus utilization per pCH we need bank-parallelism within the banks of a pCH.

However, in PIM-Potential primitives *wavesim-volume* and *wavesim-flux*, row activation accounts for 27% and 50% of total latency. This is because the benefits of bank parallelism and row locality are limited in a tightly-coupled PIM system. PIM row activation commands are generally broadcast to all banks of a pCH, hence hiding row activation delay by useful work in other banks of the same pCH is not possible, prohibiting intra-pCH bank-parallelism. Also, PIM workloads that operate on data operands residing in different DRAM rows must read data from the first row(s) into near-memory registers until it can be used in computation. Therefore, even if a workload will eventually access an entire row, the row locality that can be exploited by PIM is limited by how many elements can fit in the near-memory registers. This is particularly limiting for workloads such as *wavesim* that have multiple different operands and intermediate values from each row that must be buffered near memory. Additional measures are therefore needed to mitigate row activation overheads in tightly-coupled PIM systems.

Challenge - No benefit to data reuse: Primitives whose access patterns exhibit high cache locality are a poor fit for PIM, since on-chip reuse exploited by the GPU can outweigh the bandwidth benefit of PIM. In non-PIM systems, caches are used to automatically exploit data reuse where available. For data that is frequently accessed, memory accesses are low cost because they are likely to hit in the cache, so there is little incentive to avoid these accesses. In *ss-gemm*, the cost of operating on zero values (which don't impact the result) is minimal since data is mostly accessed from cache, so a simple dense matrix multiplication strategy is efficient. In *push-primitive*, updates to high reuse nodes and low reuse nodes are treated the same; caches will implicitly exploit reuse and maximize efficiency where available.

This is not true of PIM. Each PIM operation must occupy a memory command slot regardless of how frequently that data is accessed. Therefore, to maximize performance, a PIM system should be more judicious in issuing PIM requests to memory, avoiding accesses that aren't needed and using the GPU for accesses that exhibit high reuse.

Challenge - Command bandwidth bottleneck: We also observe in this work that PIM alters the balance of memory bandwidth demand, creating a new command bandwidth bottleneck for some workloads. In non-PIM systems, each memory command uses the exact same amount of command bandwidth to specify a command and data bandwidth to transfer data across the DRAM bus and into/out of the memory bank. Therefore, provisioning bandwidth for command and data buses is straightforward: match bandwidths to this fixed demand proportion.

However, in a tightly-coupled PIM system, demand for command and data bus bandwidth can vary based on the PIM command. All PIM commands require the use of the command bus to transmit address information, but not all commands transfer data across the DRAM data bus. For example, *push-primitive* performs in-place updates using single-bank *pim-ADD* commands (which use both command and data buses) followed by single-bank *pim-store* commands (which use the command bus only). As a result, the command

bus becomes the bandwidth limiter while the data bus goes underutilized. A rethinking of how we provision bandwidth to these channels in a PIM system is needed to provide optimal performance for any PIM workload that has a similarly mismatched command and data demand.

5 Optimized PIM

We discuss in this section methods that target the PIM-specific limitations discussed in Section 4.4 and how they impact acceleration potential.

5.1 Targeted PIM Optimizations

Some of the exposed PIM limitations motivate relatively minor hardware or software changes, while others motivate a reallocation of resources to better suit the demands of PIM-Potential workloads. We refer to all methods collectively as PIM optimizations.

5.1.1 Bank parallelism and row locality for PIM. As discussed in Section 4.4, our baseline PIM system is unable to exploit bank parallelism to hide row activation latency, and its ability to exploit row locality is limited by near memory storage capacity. Thus, activation overhead can be quite harmful to PIM performance, particularly for workloads with high PIM register storage requirements (e.g., *wavesim*). To address these limitations, we describe two targeted optimizations to the PIM system: eager row activation scheduling to better exploit bank parallelism, and enhanced register storage to better exploit row locality.

Bank parallelism via eager row activation: In our baseline PIM system, the PIM unit is shared by two DRAM banks (one odd numbered, one even numbered). While PIM column commands are multicast to one subset of banks (odd or even) in a pCH, PIM row activation commands are broadcast to all banks (odd and even) with the assumption that PIM workloads exhibit high regularity and the same row in both bank subsets is likely to be accessed. Although this amortizes the cost of the activation over more banks, this serialized latency is still exposed and can impact performance. Instead of this schedule, we propose to first split all-bank row activations into separate even and odd bank activations. This allows for a decoupled parallel schedule depicted as PIM eager-activate schedule in Figure 7a. This schedule hides activation latency behind useful work by *eagerly activating* the necessary row in one subset of banks while compute commands are being performed in the opposite subset. It also does not impact the order of compute commands or serialized activation latencies and dependencies within odd and even banks, which ensures functionally correct execution. Eager row activation commands can be generated using a compiler pass, or in hardware via augmented memory controllers.

Row locality via increased register resources: The second optimization addresses this newly exposed bottleneck in a simpler fashion: increasing near-memory register storage. By increasing the number of PIM registers available for intermediate data storage in each bank, more data can be loaded out of each activated row, better amortizing the cost of activation for that row. This optimization comes with an increased register area cost which would likely require tradeoffs in memory capacity. However, for workloads with high register pressure like *wavesim*, the performance benefits may be worth the cost.

5.1.2 Selective PIM command issue. In GPUs, selective issue of memory requests rarely makes sense for access patterns with reuse; if data is likely to be cached, the cost of determining whether to issue a request (e.g., sparse representation conversion for dynamic sparsity) may be higher than the cost of the access itself. However, this is not the case for PIM, which requires a memory command for every operation; selective request issue is more beneficial in this context.

Sparsity-aware PIM:With a sparsity-oblivious PIM design, all operations are naively offloaded to PIM, even those that do not impact the result. To ensure only useful work is processed by PIM, we propose performing additional checks (at the software level) by the GPU to *opportunistically* skip issuing PIM commands when certain conditions (e.g., sparsity) are met. Figure 7b illustrates how this works for *ss-gemm*. In the optimized orchestration, before issuing a PIM command to multiply the single X_0 value with the elements of column A, X_0 is inspected at the host. As X_0 is zero, the PIM command is not issued at all. This is possible because each PIM command multiplies a single broadcast value with all target memory elements. We term this as *sparsity-aware* PIM. Also note that this software optimization will have minimal impact on a host-only implementation because it will reduce overall memory demand only when elements in a skinny matrix row are all zero (cache reuse avoids repeated memory accesses to the same dense matrix column).

Prior work exploits sparsity in PIM operations by reducing read energy when any input zero is detected near memory [49]; sparsity-aware PIM relies on a similar principal, but it improves both performance and energy efficiency by avoiding PIM command issue entirely when a broadcast of zero value is detected in the host. Further, in contrast to existing GPU methods to exploit sparsity [55], this optimization does not rely on specialized sparsity formats which incur data-transformation and metadata overheads.

Cache-aware PIM: Similarly, when workloads exhibit variable cache reuse, statically offloading all operations to PIM leads to inefficiency for frequently reused data. To address this performance limitation, the decision of whether to offload to PIM can be made for each individual operation in a reuse-aware manner. If we can effectively predict which operands experience reuse, PIM efficiency can be improved for such workloads by *selectively* performing low-reuse operations in PIM and performing high-reuse operations in the GPU (which can take advantage of caches).

Past work has explored multiple methods for cache reuse prediction, including dynamic schemes [8] and offline schemes [4] that can be augmented to work with described PIM designs. This work does not investigate the optimal prediction policy for variable reuse workloads like *push-primitive*, but we do explore the potential benefit of such schemes by modeling perfect prediction.

5.1.3 Demand-proportional bandwidth for PIM. PIM demand for memory command and data bandwidth can vary from workload to workload, which means the fixed bandwidth ratio used by conventional memory systems may lead to underutilization. This is the case for *push-primitive*, which uses single-bank commands to read, operate on, and write elements in memory. Although *PIM-add* commands use the data bus to send source node operands, *PIM-store* commands do not. Since the data bus goes underutilized for half of the executed PIM commands, performance for this workload is

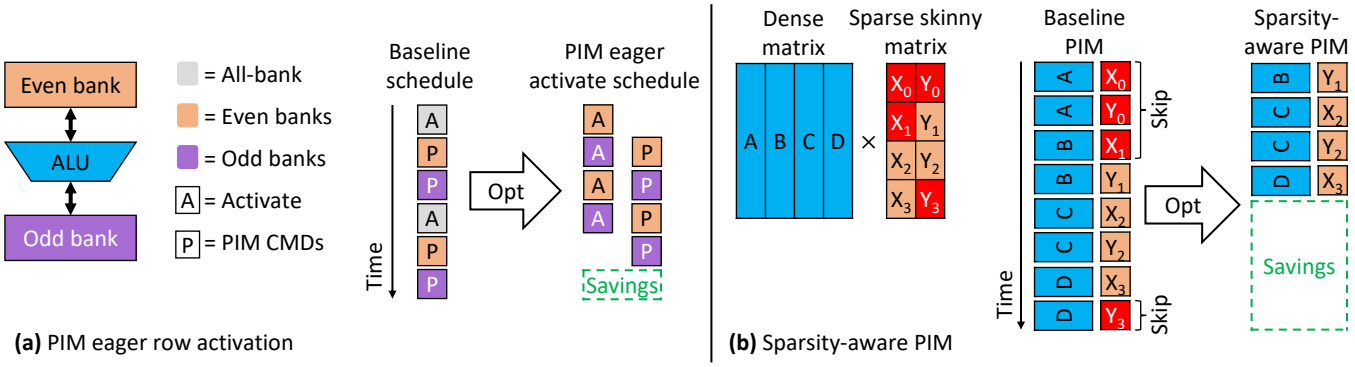

Figure 7: (a) Optimizations for improving PIM performance: eager row activation enables bank paralleism and (b) sparsity-aware PIM accesses avoid memory costs exclusive to PIM for reused data.

bottlenecked by command bandwidth. With increased command bandwidth, *push-primitive* could issue multiple concurrent store commands to disjoint banks, improving PIM performance.

There are multiple ways to increase effective command bandwidth for PIM. For in-place updates like those used in *push-primitive*, near-memory registers can be used to latch the addresses of single-bank *pim-ADD* commands and reuse them for a subsequent multi-bank *pim-store* command to the same memory locations. Alternatively, the data bus could be configured to transmit independent addresses for multiple PIM banks when not being used by data, similar to prior proposals to leverage the data bus for PIM command information [47, 48]. A third method leverages command compression to transmit more command information in a single command slot. This could involve using a single command to trigger a sequence of operations at a single bank (this bank would be blocked for scheduling until these operations completed), or it could involve leveraging commonalities in address information (e.g., if target addresses only differ in a subset of bits or by a small offset, address bits on the command bus could encode multiple target addresses relative to a previously defined base address). Finally, more bandwidth could simply be allocated to the command bus to match the demand of the target PIM workload (command bandwidth is lower and can be increased more easily than data bandwidth). All of the above cases require changes to the DRAM architecture to be able to extract/interpret the appropriate address and command bits from the DRAM interface and use these in PIM command execution. In addition, all but near-memory address latching would require changes to the DRAM interface and memory controller, as they impact how address bits are routed and/or impose additional timing restrictions on scheduling. Nevertheless, such interface flexibility may be necessary to take full advantage of PIM.

To investigate the potential impact of increased command bandwidth on command-limited workloads, we model a 4x higher bandwidth command bus (this corresponds roughly to how much more bandwidth would be available in HBM3 if the data bus were used for command transfer during store commands in *push-primitive*).

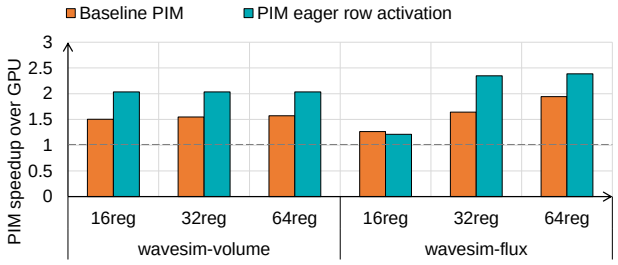

Figure 8: Optimized PIM speedup for *wavesim* primitives.

5.2 Performance Analysis

We next evaluate the implications of optimizations and techniques we discussed above on PIM performance. Optimizations are employed in a *targeted* manner, focused towards the primary bottlenecks of each primitive. Since *wavesim* primitives require storage of multiple accumulation operands in PIM registers, exacerbating activation overhead, we study eager row activation and increased register resources for these primitives. Since *ss-gemm* exhibits dynamic sparsity, we study *sparsity-aware* PIM for this primitive. Since *push-primitive* exhibits input-dependent cache locality and is limited by command bandwidth, we study *cache-aware* PIM and increased command bandwidth resources for this primitive. While these bottlenecks are specific to primitives under study, we believe they will be experienced more widely as PIM is harnessed more widely. Also note that these optimizations are complementary and can be employed in tandem in future PIM designs.

5.2.1 Wave Simulation. Wavesim acceleration improvements with eager row activation and increased register resources are depicted in Figure 8. For the *wavesim-volume* primitive, eager row activation improves PIM speedup from 1.5x to 2.04x. Further, this optimization entirely eliminates row activation overheads for this primitive such that more registers do not improve performance. This is in contrast to the *wavesim-flux* primitive which exhibits higher register pressure and row activation overheads. At lower register counts (16), *architecture-aware* activation does not improve performance because there are not enough commands per row activation to hide parallel activation latency or to amortize serial activation latency. However, more resources reduce register pressure, enabling this optimization

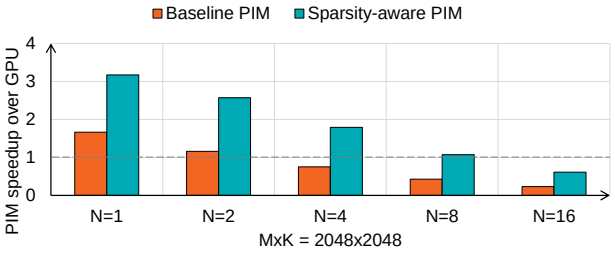

Figure 9: Optimized PIM speedup for *ss-gemm*.

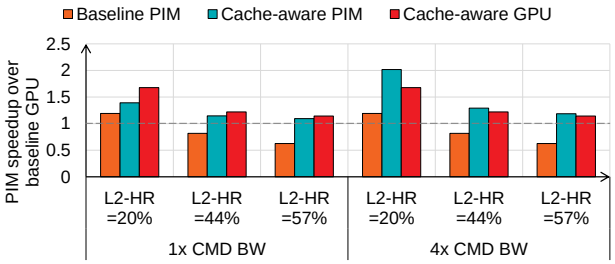

Figure 10: PIM speedup for *push-primitive* with and without cache-awareness and increased command bandwidth.

to better hide activation latency and achieve up to 2.63× speedup over the GPU baseline.

5.2.2 Sparse Skinny GEMMs. For *ss-gemm*, we focus on implications of our *sparsity-aware* PIM optimization (depicted in Figure 9). We observe here that *sparsity-aware* PIM significantly improves the PIM speedup (more than 3×) with expected tapering in benefits with increased reuse at the GPU (increasing N). Further, it also allows PIM to manifest acceleration in scenarios where baseline PIM manifested a slowdown (speedup of 1.07× for $N = 8$, while baseline PIM suffers from 57% slowdown).

5.2.3 Push-based Computation. Figure 10 shows the benefits of cache-aware PIM and increased command bandwidth for *push-primitive*. We first consider the effects of a locality-based predictor using a cache model (16-way, 4MB, LRU replacement) that classifies updates to graph nodes in *push-primitive* as either likely manifesting reuse (performed in cache) or not (performed in PIM). As before, we evaluate these workloads on three graph inputs with varying degrees of locality, each of which is labeled with its observed L2 cache hit rate. Overall, *cache-aware* PIM prevents performance degradation related to the cache reuse observed in the baseline PIM, leading to an average speedup of 1.20× (max 1.39×).

Further, we also model an optimized GPU baseline wherein the GPU can also leverage the locality predictor to reduce access granularity (i.e. use 32B rather than 64B accesses) for updates that do not benefit from caching. We term this *cache-aware GPU*, and it achieves up to 1.68× speedup relative to the baseline GPU. Although *cache-aware* PIM reduces data transferred across the memory interface relative to *cache-aware* GPU, it does not reduce the command bandwidth demand, and the PIM DRAM latency requirements actually lead to worse performance for *cache-aware* PIM.

Additional command bandwidth only benefits single-bank PIM commands that do not carry data (i.e., *push-primitive pim-store* commands). With 4× as much command bandwidth, PIM performance improves further to exceed *cache-aware* GPU performance for all inputs and provide up to 2.02× speedup relative to the baseline GPU.

Overall, there clearly is opportunity to address these emerging PIM bottlenecks and support a broader range of primitives.

6 Related Work

Many recent PIM architectures are more loosely-coupled than the high-bandwidth options studied here in that they add compute units on a "base" logic die 3D-stacked under a set of DRAM dies (e.g, hybrid memory cubes [9, 12–14, 16, 40, 53]), they add compute cores in DDR DRAM (e.g., UPMEM [20], Chameleon [10]), or they add application-specific capabilities near DDR DIMMs (e.g., TensorDIMM [30], RecNMP [26], TRiM [42]). These architectures allow software to offload coarse-grain functions to PIM, and this requires allocating significant near-memory resources for intermediate data storage and instruction fetch and sequencing. As a result, they do not suffer from the limitations discussed in this work; register pressure does not limit row locality, reused data can easily be exploited at low cost, and coarse-grain function launches are unlikely to form a command bandwidth bottleneck. However, these resource requirements also prevent loosely-coupled PIM modules from being applied at a per-bank granularity in area-constrained HBM chips, and they are unable to provide bandwidth uplift relative to a 2.5D stacked HBM memory. In addition, this asynchronous implementation motivates a programming model that uses separate memory spaces for PIM and non-PIM accesses, requiring explicit copies in and out of the PIM space (this overhead can outweigh PIM benefits when reuse is low). Therefore, they are a poor fit for accelerating memory-limited workloads on the high-throughput accelerators studied in this work.

Some PIM architectures integrate compute and memory more tightly than HBM-PIM and GDDR-PIM. These architectures attain extreme parallelism by executing bulk bitwise functions directly on bitline outputs [35, 45] or by leveraging the physical properties of non-volatile memory (NVM) to perform analog operations [36, 37, 52]. However, these systems are more limited in the types of primitives they can accelerate. While ReRAM efficiently implements dot products for weight-stationary inference and bitwise operations can be chained to implement general arithmetic, the accuracy, precision, and programmability of these systems are limited, precluding their use for many compute domains. When such architectures are leveraged for irregular workloads (e.g., GraphR [46]) the focus is energy and area efficiency rather than improved data bandwidth.

Past work has studied the role of PIM for accelerating wave simulation [24], graph analytics [7, 38, 54, 56], sparse ML [22], and many other primitives [23]. However, each of these targets a PIM architecture that differs in significant ways (they use domain-specific or loosely-coupled architectures such as UPMEM) from the tightly-coupled PIM designs studied in this work, which gives rise to differences in performance bottlenecks and potential optimizations.

7 Conclusion

To the best of our knowledge, this is the first work to evaluate tightly-coupled PIM designs across primitives from a broad set of domains. We observe that tightly-coupled PIM designs, which today are understandably geared toward ML primitives, do not provide benefits for PIM-Potential workloads (i.e., workloads that fit some but not all of the PIM-amenability characteristics). We analyze how PIM-Potential primitives can be mapped to PIM and identify bottlenecks unique to these PIM designs. Based on this analysis, we propose targeted hardware and software methods that overcome these bottlenecks, improving average PIM speedups from 1.12x to 2.49x relative to a GPU baseline. Our work demonstrates that, while emerging tightly-coupled PIM designs hold promise, to unlock broad acceleration, programmers and architects must consider the unique challenges that arise in these systems.

References

[1] Jedec high bandwidth memory (hbm) dram. https://www.jedec.org/standards-documents/docs/jesd235a, 2013.

[2] Jedec publishes hbm3 update to high bandwidth memory (hbm) standard. https://www.jedec.org/news/pressreleases/jedec-publishes-hbm3-update-high-bandwidth-memory-hbm-standard, 2022.

[3] rocprofiler developer tool. https://github.com/ROCm-Developer-Tools/rocprofiler, 2022.

[4] Abraham Addisie, Hiwot Kassa, Opeoluwa Matthews, and Valeria Bertacco. Heterogeneous memory subsystem for natural graph analytics. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 134–145. IEEE, 2018.

[5] Shaizeen Aga, Nuwan Jayasena, and Mike Ignatowski. Co-ml: a case for collaborative ml acceleration using near-data processing. In *Proceedings of the International Symposium on Memory Systems*, pages 506–517, 2019.

[6] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–492, 2017.

[7] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 105–117, 2015.

[8] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 336–348, 2015.

[9] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. *ACM SIGARCH Computer Architecture News*, 43(3S):336–348, 2015.

[10] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems. In *2016 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[11] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 93–104, 2017.

[12] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, et al. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 316–331, 2018.

[13] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T Malladi, Hongzhong Zheng, et al. Conda: Efficient cache coherence support for near-data accelerators. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 629–642, 2019.

[14] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T Malladi, Hongzhong Zheng, and Onur Mutlu. Lazypim: An efficient cache coherence mechanism for processing-in-memory. *IEEE Computer Architecture Letters*, 16(1):46–50, 2016.

[15] Preston Briggs, Keith D Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, 1994.

[16] Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R Iris Bahar. Concurrent data structures with near-data-processing: An architecture-aware implementation. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 297–308, 2019.

[17] Fred C Chow and John L Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(4):501–536, 1990.

[18] Criteo. Criteo terabyte click logs dataset. https://ailab.criteo.com/criteo-1tb-click-logs-dataset/.

[19] Timothy A Davis et al. Suitesparse: A suite of sparse matrix software, 2015.

[20] Fabrice Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24. IEEE Computer Society, 2019.

[21] Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of workshop on GRAph data management experiences and systems*, pages 1–6, 2014.

[22] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Sparsep: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–49, 2022.

[23] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system. *IEEE Access*, 10:52565–52608, 2022.

[24] Bagus Hanindhito, Ruihao Li, Dimitrios Gourounas, Arash Fathi, Karan Govil, Dimitar Trenev, Andreas Gerstlauer, and Lizy John. Wave-pim: Accelerating wave simulation using processing-in-memory. In *50th International Conference on Parallel Processing*, pages 1–11, 2021.

[25] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and TN Vijaykumar. Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 372–385. IEEE, 2020.

[26] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S Lee, et al. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 790–803. IEEE, 2020.

[27] Jin Hyun Kim, Shin-Haeng Kang, Sukhan Lee, Hyeonsu Kim, Yuhwan Ro, Seungwon Lee, David Wang, Jihyun Choi, Jinin So, YeonGon Cho, et al. Aquabolt-xl hbm2-pim, lpddr5-pim with in-memory processing, and axdimm with acceleration buffer. *IEEE Micro*, 42(3):20–30, 2022.

[28] Jin Hyun Kim, Shin-haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Bengseng Phuah, et al. Aquabolt-xl: Samsung hbm2-pim with in-memory processing for ml accelerators and beyond. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–26. IEEE, 2021.

[29] Yongkee Kwon, Kornijcuk Vladimir, Nahsung Kim, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Guhyun Kim, Byeongju An, et al. System architecture and software stack for gddr6-aim. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–25. IEEE, 2022.

[30] Youngeun Kwon, Yunjae Lee, and Minsoo 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*, pages 740–753, 2019.

[31] Chang-Chi Lee, CP Hung, Calvin Cheung, Ping-Feng Yang, Chin-Li Kao, Dao-Long Chen, Meng-Kai Shih, Chien-Lin Chang Chien, Yu-Hsiang Hsiao, Li-Chieh Chen, et al. An overview of the development of a gpu with integrated hbm on silicon interposer. In *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*, pages 1439–1444. IEEE, 2016.

[32] Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Kornijcuk Vladimir, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Jaewook Lee, Donguc Ko, Younggun Jun, Keewon Cho, Ilwoong Kim, Choungki Song, Chunseok Jeong, Daehan Kwon, Jieun Jang, Il Park, Junhyun Chun, and Joohwan Cho. A 1ynm 1.25v 8gb, 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops mac operation and various activation functions for deep-learning applications. In *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 65, pages 1–3, 2022.

[33] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 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)*, pages 43–56, 2021.

[34] Won Jun Lee, Chang Hyun Kim, Yoonah Paik, Jongsun Park, Il Park, and Seon Wook Kim. Design of processing-"inside"-memory optimized for dram behaviors. *IEEE Access*, 7:82633–82648, 2019.

[35] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. Drisa: A dram-based reconfigurable in-situ accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 288–301, 2017.

[36] Yueting Li, Tianshuo Bai, Xinyi Xu, Yundong Zhang, Bi Wu, Hao Cai, Biao Pan, and Weisheng Zhao. A survey of mram-centric computing: From near memory to in memory. *IEEE Transactions on Emerging Topics in Computing*, 2022.

[37] Sparsh Mittal. A survey of reram-based architectures for processing-in-memory and neural networks. *Machine learning and knowledge extraction*, 1(1):75–114, 2018.

[38] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International symposium on high performance computer architecture (HPCA)*, pages 457–468. IEEE, 2017.

[39] Lifeng Nai, Yinglong Xia, Ilie G Tanase, Hyesoon Kim, and Ching-Yung Lin. Graphbig: understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[40] Ravi Nair, Samuel F Antao, Carlo Bertolli, Pradip Bose, Jose R Brunheroto, Tong Chen, C-Y Cher, Carlos HA Costa, Jun Doi, Constantinos Evangelinos, et al. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3):17–1, 2015.

[41] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[42] Jaehyun Park, Byeongho Kim, Sungmin Yun, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. Trim: Enhancing processor-memory interfaces with scalable tensor reduction in memory. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 268–281, 2021.

[43] Suchita Pati, Shaizeen Aga, Nuwan Jayasena, and Matthew D. Sinclair. Demystifying bert: System design implications. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 296–309, 2022.

[44] Giordano Salvador, Wesley H Darvin, Muhammad Huzaifa, Johnathan Alsop, Matthew D Sinclair, and Sarita V Adve. Specializing coherence, consistency, and push/pull for gpu graph analytics. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 123–125. IEEE, 2020.

[45] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 273–287, 2017.

[46] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Graphr: Accelerating graph processing using reram. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 531–543. IEEE, 2018.

[47] Chirag Sudarshan. *Processing-in-Memory DRAM Architectures for Neural Network Applications*. PhD thesis, Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau, 2024.

[48] Chirag Sudarshan, Mohammad Hassani Sadi, Lukas Steiner, Christian Weis, and Norbert Wehn. A critical assessment of dram-pim architectures-trends, challenges and solutions. In *International Conference on Embedded Computer Systems*, pages 362–379. Springer, 2022.

[49] Chirag Sudarshan, Mohammad Hassani Sadi, Christian Weis, and Norbert Wehn. Optimization of dram based pim architecture for energy-efficient deep neural network training. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1472–1476. IEEE, 2022.

[50] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. Gunrock: Gpu graph analytics. *ACM Transactions on Parallel Computing (TOPC)*, 4(1):1–49, 2017.

[51] Lucas C Wilcox, Georg Stadler, Carsten Burstedde, and Omar Ghattas. A high-order discontinuous galerkin method for wave propagation through coupled elastic–acoustic media. *Journal of Computational Physics*, 229(24):9373–9396, 2010.

[52] Yue Xi, Bin Gao, Jianshi Tang, An Chen, Meng-Fan Chang, Xiaobo Sharon Hu, Jan Van Der Spiegel, He Qian, and Huaqiang Wu. In-memory learning with analog resistive switching memory: A review and perspective. *Proceedings of the IEEE*, 109(1):14–42, 2020.

[53] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 85–98, 2014.

[54] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 544–557. IEEE, 2018.

[55] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 359–371, 2019.

[56] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. Graphq: Scalable pim-based graph processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 712–725, 2019.