2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)

Concurrent PIM and Load/Store Servicing in PIM-Enabled Memory

Sudhanshu Gupta*, Niti Madan†, Sooraj Puthoor†, Nuwan Jayasena†, and Sandhya Dwarkadas‡
*Department of Computer Science, University of Rochester
†AMD Research and Advanced Development
‡Department of Computer Science, University of Virginia
Email: *sgupta45@cs.rochester.edu, † {firstname.lastname}@amd.com, ‡sandhya@virginia.edu

***Abstract*—Processing in-memory (PIM) has emerged as a promising approach to address the increasingly memory bound nature of modern applications like machine learning and genomics. While PIM-enabled memories offer significant performance and energy improvements over host-side execution, integration of such memories into existing systems remains an open challenge. In particular, naively replacing regular memory with a PIM-enabled one in a conventional processor could be detrimental to its performance. PIM applications are optimized to saturate the memory subsystem to maximize speedup. However, since modern processors, including CPUs and GPUs, support multi-tenancy to improve utilization, such saturation can lead to extreme unfairness and denial of service to other applications.**

In this paper, we characterize the performance of a PIM-enabled GPU system when co-executing regular GPU kernels with a PIM kernel. Our characterization shows that PIM kernels can easily overwhelm the interconnect and the memory controller and severely degrade the performance of the non-PIM kernel, hurting system-level fairness and throughput metrics. Based on this characterization, we propose changes to the interconnect that ease the flow of requests from the processor to the memory controller. At the memory controller, we propose a new scheduling policy, called F3FS, that optimizes for fairness and throughput. While F3FS benefits from changes to the interconnect, we show that it performs comparably to existing policies without them. We evaluate and compare the proposed changes to state-of-the-art memory controller scheduling policies under both competitive (two kernels from different applications) and collaborative (two kernels from same application) scenarios.

***Index Terms*—General-purpose graphics processing unit, interconnect, memory access scheduling, processing in-memory.**

I. Introduction

Modern applications, ranging from consumer mobile applications [9] to large server workloads [17], [65], are becoming increasingly memory bound. While newer memory technologies like HBM [28], [33], GDDR6 [29], and HMC [20], [27] reduce the memory bottleneck by offering wider links, increased parallelism, and improved scalability [11], they still struggle to close the gap between processor and memory performance, the so-called ”memory wall” [23], [67].

Processing in-memory (PIM) [1], [18], [19], [22], [42], [45], [63] is a paradigm shift in how we design our computers, dictating that compute be moved closer to data instead of the other way around. PIM architectures place compute units close to/inside main memory cells, minimizing data movement costs and achieving wide data parallelism. A common mechanism to trigger computation on these compute units is by submitting PIM requests, requests that resemble regular memory requests (henceforth called MEM requests) but perform computation in-place rather than moving data to/from the host processor. To handle this heterogeneity in request types, the memory controller needs to switch between *MEM mode* and *PIM mode* to service requests of each type. This switching adds a new dimension to memory controller scheduling that is distinct from traditional architectures.

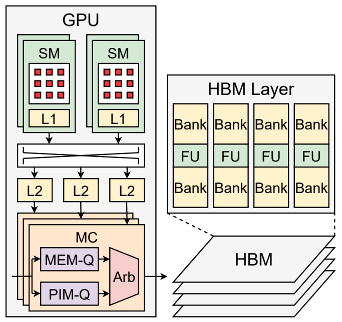


Fig. 1: PIM-enabled GPU architecture. Each HBM layer contains eight functional units (FUs) that are shared by two banks each (fewer shown in figure for readability).

Though PIM-enabled memories can be paired with many host processors, GPUs have emerged as a particularly attractive fit given their highly data-parallel architecture [1], [22], [54]. Figure 1 shows an example PIM-enabled GPU architecture. Modern GPUs often utilize techniques like streams [4], [21], [50], multi-process service (MPS) [52], and multi-instance GPU (MIG) [51] to execute multiple kernels concurrently, allowing for improved utilization of the GPU's resources. These techniques enable simultaneous use of both GPU cores and PIM functional units (FUs) by allowing for co-execution of traditional GPU kernels with kernels that submit PIM requests to the main memory for computation [1], [54] (henceforth referred to as GPU and PIM kernels, respectively), improving both resource utilization and application performance. Such simultaneous use can be either *collaborative*, where both GPU and PIM kernels belong to the same application (e.g., large language models [24], [53], [57], graphics [68], and scientific computing [25]), or *competitive*, where two separate applications launch kernels on the two resources [13]. In both cases, the resulting simultaneous use of memory by both GPU and PIM kernels raises an important question: *how do we efficiently route and schedule MEM and PIM requests to ensure fairness between co-executing kernels while also maximizing system throughput?*

To answer this question, we need to look at the two key resources that are shared by MEM and PIM requests: 1) the memory controller, and 2) the interconnect between the GPU's streaming multiprocessors (SMs) and the memory controllers. We first discuss contention at the **memory controller**. Figure 1 shows how the memory controller (MC) maintains separate queues for MEM and PIM requests, with an arbiter (Arb) to switch between them. The switching policy has a direct impact on system efficiency since switching modes: 1) requires the draining of in-flight requests, potentially causing bank idle time, and 2) can hurt locality since MEM and PIM requests often map to different rows. These factors influence queueing delay and the rate at which each request type is served, thereby directly affecting the performance of both GPU and PIM kernels. Optimizing for both system throughput and application fairness is a hard problem, since throughput favors infrequent switching while fairness favors frequent switching. This requires the design of a smart switching policy that can balance the two goals.

PIM kernels are optimized to fully utilize SM resources to send as many PIM requests as possible. Since the memory controller may not be able to keep up with this burst of requests under contention, PIM requests can quickly fill up the memory controller queues and create backpressure in the **interconnect**, causing denial of service to MEM requests. Not only can this stall the SMs executing GPU kernels, but also reduce the memory controller's visibility into the load/store stream and lead to poorer decision making.

Motivated by these challenges, this paper makes the following contributions:

- A comprehensive analysis and characterization of GPU/PIM co-execution on a PIM-enabled GPU under 180 competitive scenarios and a GPT-3-like collaborative scenario, focused on interconnect and memory controller bottlenecks. We discover that PIM kernels can easily overwhelm the shared memory subsystem by its high request injection rate, causing unfairness. Concurrently, inefficient switching at the memory controller can further exacerbate fairness and throughput bottlenecks.
- Evaluation of the efficacy of adding a separate virtual channel (VC) for PIM requests to alleviate congestion at the interconnect. Our analysis shows that this can improve the arrival rate of MEM requests at the memory controller by an average of 2.87x for some memory controller scheduling policies, while adding less than 5% area overheads.
- Design and evaluation of a novel memory controller scheduling policy, called F3FS, that modifies FR-FCFS by adding an extra layer of arbitration to favor current mode, but caps the number of each request type served to provide fairness. The cap can be adjusted to provide fairness between competing applications or reduce execution time for collaborating ones.

II. Background

A. PIM Architectures

The placement of functional units in/near memory determines the efficiency of the resulting PIM architecture. Cost and performance increase in going from architectures that place compute between the memory chips and the host interface, (such as on the base die of HBM [3], [13], [35], [54]) to subarray-level PIM, where compute is placed at each subarray within a bank [66]. On this spectrum of performance and complexity, bank-level PIM has emerged as a practical middle ground by placing compute near the banks within the main memory [1], [22], [38], [40], [42], [60], offering reduced execution time and energy consumption [1]. Figure 1 shows such an architecture.

Figure 2 shows the microarchitecture of a PIM functional unit (FU). Each FU incorporates a SIMD ALU along with a local register file to store operands and temporary values. The register file is DRAM word-wide, which is typically tens of bytes (32 bytes in HBM). The SIMD ALU operates on a DRAM word, which can include multiple data elements (e.g., 16 FP16 elements [22], [42]).

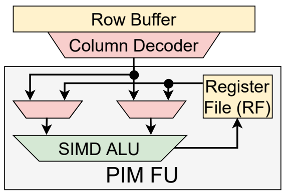


Fig. 2: PIM functional unit (FU) microarchitecture. The SIMD ALU can implement generic math and logic operations and/or domain specific operations.

PIM kernels mapped to bank-level PIM architectures lay out data in row buffer-sized chunks across multiple banks, processing them in parallel one DRAM word at a time. Given the large datasets of the applications that PIM targets, this could result in a large influx of requests in a very short period. In order to curb this injection rate and provide higher command bandwidth, bank-level PIM architectures often implement a separate PIM mode [22], [40], [42], [60]. Within PIM mode, a single PIM request is executed by all banks in a lock-step manner. The memory controller switches between PIM and MEM modes, effectively choosing which of PIM queue and MEM queue to service requests from (Figure 1). The PIM register file (RF) holds state across MEM/PIM switch boundaries, allowing for PIM correctness.

B. PIM Programming

There are two broad PIM programming paradigms: 1) coarse-grained offloading, where the application configures control registers in the memory controller that specify the function to compute [13], [16], and 2) fine-grained offloading, where the application issues special memory instructions that encode PIM operations (e.g., add), which are scheduled by the memory controller [41], [60]. The fine-grained instructions look and behave like non-temporal (i.e., non-cached) stores for the host core, and the model we use in this paper.

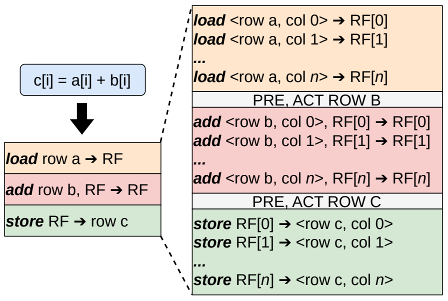

c[i] = a[i] + b[i]
load row a ➔ RF
add row b, RF ➔ RF
store RF ➔ row c
load <row a, col 0> ➔ RF[0]
load <row a, col 1> ➔ RF[1]
...
load <row a, col *n*> ➔ RF[*n*]
PRE, ACT ROW B
add <row b, col 0>, RF[0] ➔ RF[0]
add <row b, col 1>, RF[1] ➔ RF[1]
...
add <row b, col *n*>, RF[*n*] ➔ RF[*n*]
PRE, ACT ROW C
store RF[0] ➔ <row c, col 0>
store RF[1] ➔ <row c, col 1>
...
store RF[*n*] ➔ <row c, col *n*>

Fig. 3: Vector addition PIM kernel. PIM kernels have a block structure, where a block consists of consecutive PIM operations to the same row. The size of the block is usually a multiple of the register file (RF) size (*n*).

Figure 3 shows an example fine-grained PIM kernel that adds two vectors. The vectors are laid out in separate rows and aligned to the row buffer size. The kernel first loads *n* DRAM word-sized chunks of vector ***a*** into the PIM register file. The register file contents are then added to vector ***b***, performing a DRAM word-wide SIMD operation and storing the sum into the register file. Finally, the register file contents are stored into vector ***c***.

The figure also exemplifies the *block* structure of PIM kernels, where blocks consist of consecutive PIM operations (e.g., *load*) to the same row and are separated by a precharge and an activate. While instructions within blocks can be reordered, blocks must be executed sequentially for correctness due to their dependencies. Such sequential ordering can be achieved on the host side by using special barriers (e.g., Orderlight [48], which prevents reordering at SM's operand collector stage) and at the memory controller by using a scheduling policy like first-come first-served. We use this block structure to minimize MEM interference with PIM (Section VII-B).

C. Concurrent GPU Kernel Execution

Concurrent utilization of host and PIM cores is an effective way of maximizing hardware utilization and application performance. Large language models (LLMs) exploit parallelism by simultaneously computing on different fully connected layers on the host and PIM [57] and by overlapping Query/Key/Value (QKV) generation on the host with multi-head attention on PIM [24], [53]. Other domains, like graphics [68] and scientific computing [25] also achieve performance and energy gains with such concurrent execution.

While streams [4], [21], [50] can be used to launch concurrent requests from the same application, CUDA multi-process service (MPS) [52] allows for GPU resources to be shared simultaneously by different host processes. Kernels from different processes each have their own address space, but they share the GPU SMs, caches, and memory bandwidth. Taking a step further, CUDA multi-instance GPU (MIG) [51] adds the ability to *physically* partition GPU resources (including SMs, interconnect links, memory capacity, and memory bandwidth) into several *instances*, essentially creating several sub-GPUs that can be used by independent applications.

III. Evaluation Methodology

A. Simulator

We use a modified version of GPGPU-Sim [34] that implements a cycle-level all-bank PIM execution model, closely based on commercial designs [42]. Table I summarizes key architectural parameters. The main memory incorporates a PIM FU for a pair of banks, with each bank receiving 8 register file entries out of 16. The memory controller is updated to incorporate separate MEM and PIM queues. PIM kernels, implemented in CUDA following the ISA of the PIM architecture we model [42], send PIM operations modeled as cache streaming (CS) stores[1]. We modified GPGPU-Sim's core model to ensure that CS stores bypass all caches and are sent to the main memory directly.

TABLE I: Simulation Parameters

GPU Parameters	
GPU Model: Nvidia Quadro GV100	
Number of SMs: 80	Core Frequency: 1132 MHz
L1D Cache: 32 KB	Shared Memory: 96 KB
L1I Cache: 128 KB	L2 Cache: 6 MB
Memory Parameters	
Memory Technology: HBM	
Channels/Banks: 32/16	DRAM Frequency: 850 MHz
Bus Width: 16 B	Burst Length: 2
MEM-Q/PIM-Q Size: 64 entries	NoC buffer size: 512 entries
PIM FUs: 8/channel	PIM RF Size: 16 entries
Timing Parameters (cycles):	*tCCDs*=1, *tCCDl*=2, *tRRD*=3, *tRCD*=12, *tRP*=12 *tRAS*=28, *tCL*=12, *tWL*=2, *tWR*=10, *tRTPL*=3
Address Map (bits):	RRRR.RRRRRRRR.RBBBCCCB.DDDDDCCC Key: R=Row, B=Bank, C=Column, D=Channel

B. Benchmarks

We evaluate on two application scenarios: competitive, where two separate applications launch a GPU and PIM kernel, and collaborative, where the same application launches both. The GPU and PIM kernels are launched concurrently using CUDA streams.

In order to facilitate PIM programming, we turned off pseudo-random I-poly [55] mapping to channels in favor of a more regular scheme, listed in Table I. PIM kernels use the simplified mapping to map each warp to a single memory channel and each thread within a warp to a single bank. This mapping ensures that requests to each PIM unit are issued sequentially. We use Orderlight barriers [48] to prevent reordering of requests within the SM. PIM kernels require eight SMs (total of 1024 threads, 4 warps per SM) to maximize speedup, leaving 72 SMs for the GPU kernel.

[1]https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#cache-operators

Competitive: We borrow nine PIM-amenable kernels from prior work [48][2]. Table III lists the name and input size of each benchmark. These PIM kernels are run concurrently with 20 Rodinia benchmarks [12][3] on the host GPU, giving us 180 unique GPU/PIM kernel combinations. Input size for the GPU benchmarks, listed in Table II, is taken from prior work [26], [30]. Both PIM and GPU kernels are run continuously in a loop until each kernel has run at least once. We report data for the first completed run of each benchmark.

TABLE II: GPU Benchmarks

No.	Benchmark	Input size
G1	b+tree	1 million keys, 10000 bundled queries, a range search of 6000 bundled queries with the range of each search 3000
G2	backprop	655360 input nodes
G3	bfs	1 million vertices
G4	cfd	97K elements
G5	dwt2d	1024x1024 images, forward 5/3 transform
G6	gaussian	2048x2048 matrix
G7	heartwall	656x744 video, 2 frames
G8	hotspot	2048x2048 data points, pyramid height=4, 2 iterations
G9	hotspot3D	512x512 data points, 8 layers, 10 iterations
G10	huffman	262144 elements
G11	kmeans	494020 points, 34 features
G12	lavaMD	1000 boxes
G13	lud	2048x2048 data points
G14	mummergpu	Reference: 20K sequences, 71 characters; Query: 50K sequences, 25 characters
G15	nn	10000390 hurricanes across 10 files, 10 nearest neighbors
G16	nw	2048x2048 data points
G17	pathfinder	100000x100 grid, pyramid height=4
G18	srad_v1	512x512 data points, 100 iterations, lambda=0.5
G19	srad_v2	2048x2048 data points, 2 iterations, lambda=0.5
G20	streamcluster	65536 points, 256 dimensions, 10-20 centers, 1000 intermediate centers

TABLE III: PIM Benchmarks

No.	Benchmark	Input size
P1	Stream Add	67M elements per vector
P2	Stream Copy	
P3	Stream Daxpy	
P4	Stream Scale	
P5	BN Fwd	8M batches, with 8 elements each
P6	BN Bwd	
P7	Fully connected	Inputs = Outputs = 16, 262,144 batches
P8	KMeans	1,048,576 points, 32 features
P9	GRIM	8M bitvectors, 32 base pairs

Collaborative: We emulate the execution of a GPT-3-6.7B like large language model (LLM) [10], overlapping the execution of QKV generation with multi-head attention (MHA), similar to prior work [53]. To achieve this, we execute three GEMM kernels in a series on the GPU (QKV generation) with the PIM executing GEMV and softmax layers (MHA). The model uses a batch size of 128, sequence length of 1024, and an embedding table of size 4096. We assume that the KV cache for each layer is loaded on demand to keep the model memory footprint in check.

[2]STREAM-Triad was excluded because it has the same access pattern as STREAM-Add. Histogram was excluded because only a small fraction is PIM-amenable.

[3]We do not evaluate `leukocyte` because the provided input file did not generate significant memory traffic. `hybridsort` and `particlefilter` ran for too long. `myocyte` encountered a bug with GPGPU-Sim.

C. Metrics

Competitive: We compute the speedup of both the GPU and PIM kernels as the ratio of their execution time when run alone on 80 SMs and 8 SMs, respectively, to their execution time when run under contention. We then use this speedup to evaluate each scheduling policy across two key metrics: fairness and throughput. Fairness is defined using Fairness Index [15] which quantifies the disparity between the individual GPU and PIM kernel speedups. It is expressed as:

$$Fairness\ Index = min(\frac{Speedup_{PIM}}{Speedup_{MEM}}, \frac{Speedup_{MEM}}{Speedup_{PIM}}) \quad (1)$$

Throughput is defined using System Throughput [15] which quantifies the kernel execution rate of the system, measured as the sum of speedups of the GPU and PIM kernels. This is a direct measure of concurrency and the rate at which the system can service kernels.

Collaborative: The key metric in this scenario is the speedup of the concurrent kernel execution relative to the serial execution of the kernels. We compare this speedup to an ideal scenario where the total execution time is the execution time of the longer running kernel when run alone, representing perfect overlap.

D. Memory Controller Scheduling Policies

We summarize below the baseline memory scheduling policies we evaluate. Note that most of these policies were not designed for PIM architectures; we therefore explain how they switch between PIM and MEM modes.

1) *First-Come First-Served (FCFS)*: Executes requests in the order they arrive. Switches modes according to the request type.
2) *MEM-First*: Always issues MEM requests, if there are any. Prior art has used this policy before [13].
3) *PIM-First*: Always issues PIM requests, if there are any.
4) *First-Ready FCFS (FR-FCFS)* [56]: Prioritizes row buffer hits over the oldest request, switching modes if the oldest request is from a different mode at the time of a row buffer conflict on all banks. Each bank maintains a conflict bit, which is set when there is a row buffer conflict and the oldest request is from a different mode. The bank then stalls until a mode switch occurs, which is performed after every bank has set its conflict bit.
5) *FR-FCFS-Cap:* A fairer version of FR-FCFS that CAPs the number of row buffer hits that bypass the oldest request [46].
6) *Blacklisting Memory Scheduler (BLISS)* [62]: Blacklists applications that issue more than n requests consecutively under FR-FCFS. Then implements the following priority order: 1) non-blacklisted application first, 2) row

buffer hit first, 3) oldest first. The blacklist is cleared every few thousand cycles. This mechanism effectively deprioritizes high memory intensity applications.

7) *First-Ready Round-Robin FCFS (FR-RR-FCFS)* [31]: Modifies FR-FCFS to improve fairness by cycling through modes on row buffer conflicts, effectively implementing the following priority order: 1) row buffer hit first, 2) next mode in round-robin order first, 3) oldest first within the current mode.
8) *Gather & Issue (G&I)* [41]: Switches to PIM when PIM queue occupancy reaches a *high* watermark, then drains the queue until the occupancy falls below a *low* watermark.

Each of the above described policies use FR-FCFS within MEM mode, except FCFS, while PIM requests always execute in FCFS order to ensure correctness.

IV. Characterizing GPU/PIM Interference

In order to understand the performance impact of co-executing PIM and GPU kernels, we first quantify each application type's memory behavior. Figure 4 compares the memory access characteristics of the Rodinia benchmark suite to the PIM kernels under FR-FCFS policy, in terms of (a) interconnect request arrival rate, (b) DRAM request arrival rate, (c) DRAM bank-level parallelism (BLP), and (d) DRAM row buffer hit-rate (RBHR). The boxes represent the inter-quartile range for each metric, with the middle line and whiskers representing the median and extremes, respectively. Request arrival rates are measured in terms of total GPU cycles, while BLP is measured in terms of active DRAM cycles, i.e., the average BLP while the DRAM is servicing at least one request. Since PIM kernels only need eight SMs to fully saturate the memory subsystem interface, we compare them to Rodinia kernels running on both 80 and 8 SMs (represented as GPU-80 and GPU-8, respectively).

PIM kernels have a 3.95x higher request arrival rate into the interconnect compared to GPU-8, and is only 17.8% lower than GPU-80, on average. While regular memory requests get filtered by the L2 cache, PIM requests are not, worsening the imbalance at the memory controller. PIM request arrival rate at the memory controller is heavier than both GPU-8 *and* GPU-80, on average, outpacing them by 8.33x and 2.07x, respectively.

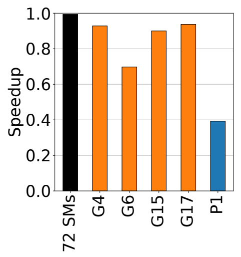


Fig. 5: Average speedup of Rodinia benchmark suite when running on 72 SMs and when co-executing with four memory intensive kernels.

Not only can MEM and PIM requests not be issued concurrently, but they also exhibit very different memory access behavior. Figures 4c and 4d compare the BLP and the RBHR of the GPU and PIM kernels. PIM kernels not only execute on all banks at the same time (Figure 4c shows a single bar at 16 for PIM kernels), but

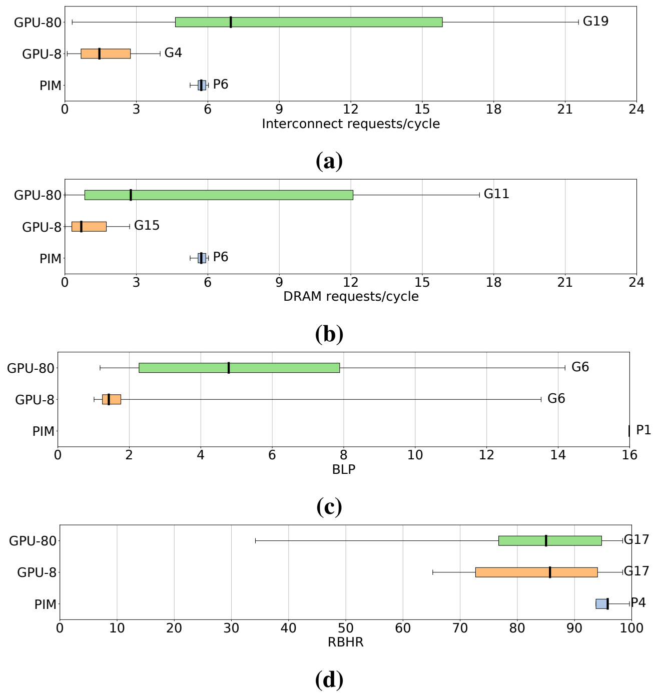
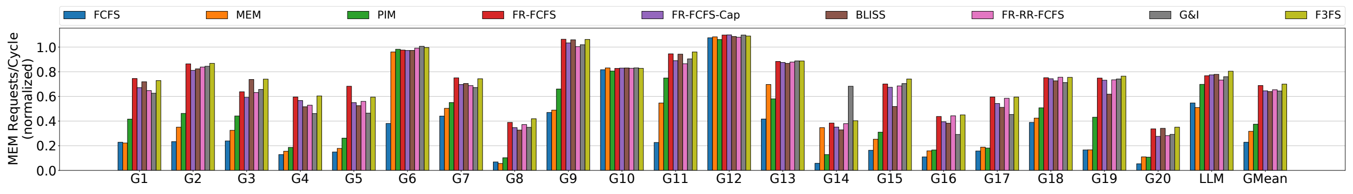


Fig. 4: Memory access characteristics of the Rodinia benchmark suite, running on 80 and 8 SMs, and the PIM kernels in terms of (a) interconnect request arrival rate, (b) DRAM request arrival rate, (c) DRAM bank-level parallelism (BLP), and (d) DRAM row buffer hit rate (RBHR). The high whiskers are labeled with the most intensive kernel for that metric.

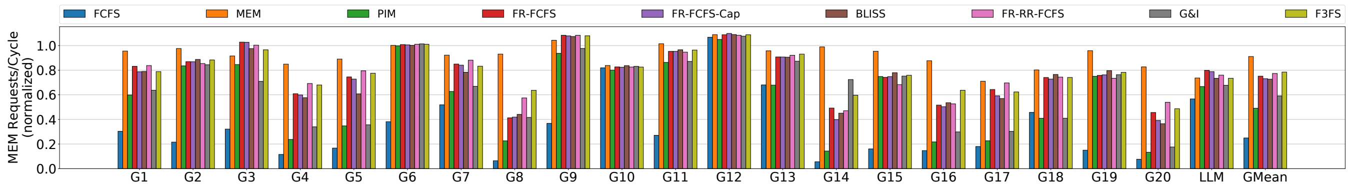
also exhibit high row buffer locality. Combined with their high request arrival rate (Figure 4b), PIM kernels can severely affect a co-executing application's performance. Figure 5 compares the impact of memory intensive GPU kernels and PIM kernels on co-executing kernels. The figure shows the average speedup of the Rodinia benchmark suite running on 72 SMs, with the remaining 8 SMs occupied by one of G4, G6, G15, G17, and P1. The four chosen GPU kernels are the most memory intensive in terms of interconnect requests (G4), DRAM requests (G15), BLP (G6), and RBHR (G17) when running on 8 SMs (Figure 4c). PIM kernels show very little variation across each metric and so we picked P1. The speedup is normalized to Rodinia benchmark suite running alone on 80 SMs. To separate the effects of memory contention and reduced SM availability, the figure also presents the speedup of running the kernels on 72 SMs without any contention. The figure shows how the benchmark suite slows down by an average of 60% when co-executing with P1, compared to a worst-case average slowdown of 30% when running with other Rodinia kernels.

V. Memory Access: Interconnect Bottlenecks

PIM kernels are optimized to maximize the utilization of PIM FUs by saturating the memory subsystem. This leads to a very high request arrival rate for the duration of the kernel's execution that can deny service to co-executing applications



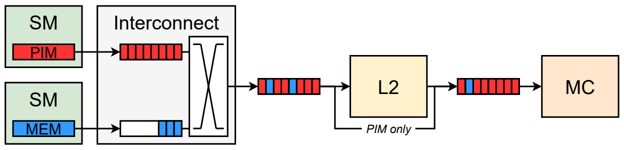
(a) Single virtual channel.



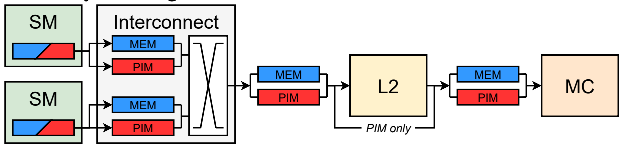
(b) Separate virtual channels.

Fig. 6: MEM request arrival rate into the memory controller, without (a) and with (b) separate MEM and PIM virtual channels, normalized to standalone execution (higher is better).

(Figure 4a). Figure 7a shows this scenario, where the PIM requests fill the interconnect→L2 and L2→DRAM queues, denying service to GPU kernels. To quantify this degradation, Figure 6a characterizes the request arrival rate of each GPU kernel under each memory scheduling policy, averaged across all PIM kernels. We present results for each scheduling policy since the service rate of each policy determines how fast the PIM requests are drained from the interconnect. Note that some applications experience an increase in the arrival rate. This is because PIM interference increases MEM queuing delay, improving MEM RBHR and reducing GPU kernel's overall execution time.



(a) PIM kernels have very high request arrival rates, causing congestion at the interconnect, interconnect→L2, and L2→DRAM queues, and unfairly slowing down GPU kernels.



(b) Separation of MEM (blue) and PIM (red) requests into separate virtual channels and queues to minimize interference between them.

Fig. 7: Comparison of the baseline memory subsystem (a) with our proposed changes (b).

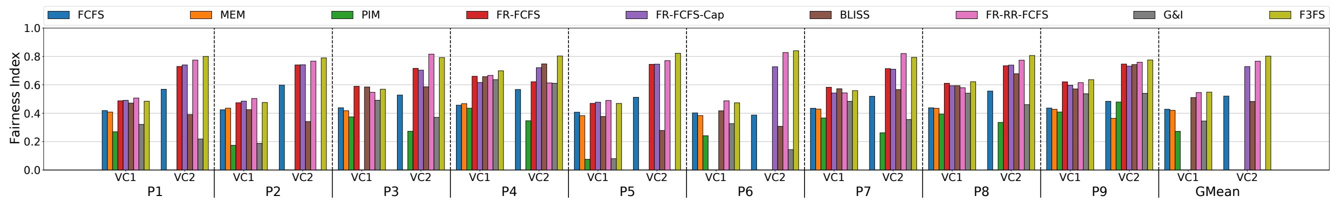
While throughput optimizing policies like FR-FCFS are able to sustain a higher arrival rate for MEM requests than others, the degradation remains severe, with even FR-FCFS suffering a 41% drop on average. A policy like MEM-First should, intuitively, perform well here, but its performance is limited by the fact that most MEM requests are stalled behind PIM requests in the interconnect. This demonstrates that even though the memory controller scheduling policy impacts interconnect congestion, PIM kernels' memory intensity necessitates changes to the interconnect architecture.

A. *Separating MEM and PIM Virtual Channels*

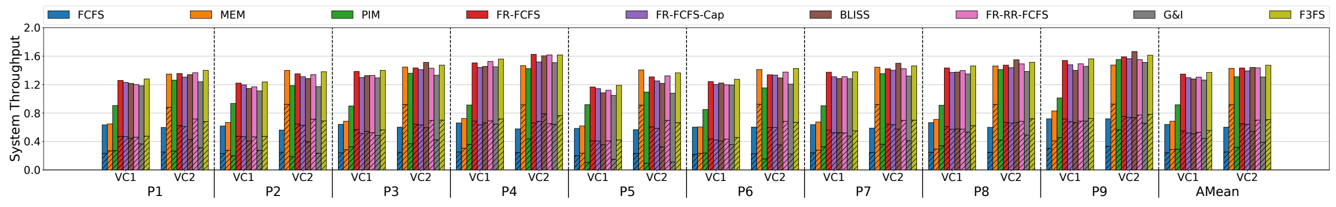
In order to alleviate the problem of congestion at the interconnect, we propose separating MEM and PIM requests into separate queues all the way from the SMs to the memory controller. Figure 7b illustrates this proposal. Memory requests entering the interconnect from the SMs are split into two virtual channels (VCs), one each for MEM and PIM requests. These virtual channels empty into interconnect→L2 cache queues, where MEM requests are picked up by the cache while PIM requests are forwarded to the memory controller. Finally, the two request types also share the links between L2 cache and memory controllers, necessitating splitting of L2→DRAM queues as well.

Using separate VCs and queues ensures that the two request types do not interfere until they reach the memory controller, preventing PIM requests from stalling MEM requests. Furthermore, the system provides fairness at each link by switching between the two queues in a round-robin fashion. The crossbar interconnect uses a modified version of the iSlip algorithm [44] where the arbiter records the previous VC served for each incoming link and switches to the other VC presuming there is traffic on it. The efficacy of this solution is demonstrated in Figure 6b. We split existing interconnect queues in half to add a PIM VC, keeping the *total* queue size in Figures 6a and 6b equal. While most policies experience an increase in the arrival rate, MEM-First experiences the biggest jump, with its average degradation reducing from 68% to 9% (2.87x improvement).

Adding virtual channels entails area and power costs, however. The VC allocator, used for allocating output virtual channels to input virtual channels, grows quadratically in the number of ports and virtual channels [69]. Meanwhile, the number of control wires to encode the VC information with each packet grows logarithmically. Despite the significant



(a) Fairness Index (FI). FI of 0 indicates starvation.



(b) System Throughput. The shaded and non-shaded regions represent MEM and PIM speedups, respectively.

Fig. 8: Fairness (a) and throughput (b).

asymptotic growth, additional VCs add modest area and power overheads, especially if the queues are long and the routers are pipelined. Based on the data from Yoon et al. (Figure 4(c) in [69]), a router based on 45nm technology with 32 queue entries, 128-bit channels, and a clock speed of 1ns experiences ∼5% increase in area when going from a single VC to two. Since our evaluated system is based on a smaller process node (12nm), uses longer queues (512 entries), and runs at a comparable clock speed (0.88ns), we expect the overheads to be even lower.

VI. Memory Utilization: Scheduling Bottlenecks

Past the interconnect queues, MEM and PIM requests again contend at the memory controller. The memory controller needs a *mode switching policy* to switch between MEM and PIM modes to serve requests of each type. The design of an efficient mode switching policy is non-trivial, for two reasons. The first reason is the increase in queuing delays. Since MEM and PIM requests cannot execute concurrently, each of them suffers increased queueing delays while waiting for requests of the other type to complete. However, each application has a different tolerance for queueing delays, creating a fairness problem.

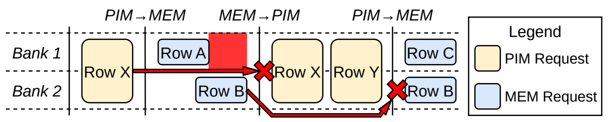


Fig. 9: Switching between MEM and PIM modes leads to loss in locality since the two request types often map to different rows, as seen for requests mapping to Rows X (PIM) and B (MEM). MEM→PIM switches also suffer from bank idle time, like Bank 1 in the figure, since MEM requests on different banks execute asynchronously.

The second difficulty is the non-trivial cost of switching, as depicted in Figure 9. When the memory controller performs a MEM→PIM switch, all in-flight MEM requests must be drained before any PIM request can be issued. Since each memory bank services requests concurrently and independently, this leads to idle time for banks that finish first (Bank 1 in Figure 9). In addition, both MEM→PIM and PIM→MEM switches may cause reduced row buffer locality since the two types of requests often map to different rows (Rows X and B in Figure 9).

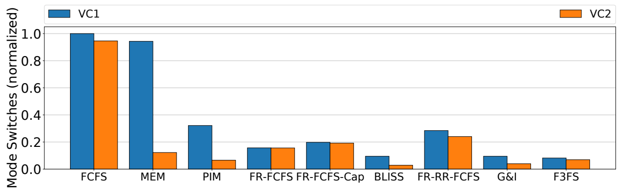
A. Competitive Co-execution

We first characterize the performance of the various memory controller scheduling policies that we listed in Section III-D under the competitive scenario, both with and without separate MEM/PIM VCs. We label the two configurations as VC1 (Figure 7a) and VC2 (Figure 7b). Figure 8 shows the fairness index and throughput for each PIM kernel, averaged across all GPU kernels. Figure 10 presents the average (a) number of mode switches (normalized to FCFS), (b) number of additional MEM conflicts per switch, and (c) latency of draining MEM queue per switch, across all combinations. Figure 10a uses geometric mean, while 10b and 10c use arithmetic mean. The figures also include results for our proposed policy, which we will introduce and discuss later in Section VII.

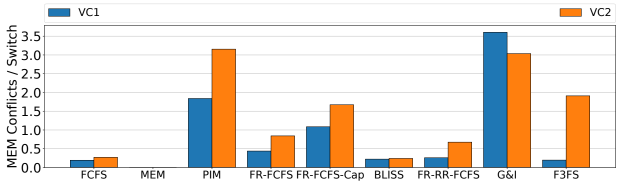
FCFS schedules memory requests in arrival order, resulting in frequent switches (Figure 10a). As a result, both individual application performance and hardware utilization can be compromised. While such switching helps fairness to a certain degree, especially with the VC2 configuration, the lack of locality and parallelism awareness hurts throughput.

MEM-First and **PIM-First** favor a single request type, with the potential for the other request type to experience extreme unfairness or starvation: a fairness index of 0 is common (Figure 8a). Most throughput gains often stem from a single application that submits the request type favored by the policy (Figure 8b). Both policies also suffer from frequent switching (Figure 10a) and high switch overheads (Figures 10b and 10c), particularly in the VC1 configuration.

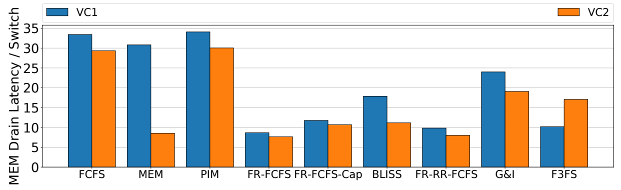
FR-FCFS optimizes for locality by prioritizing row buffer hits over older requests. Such a design introduces two sources of unfairness: 1) high row buffer locality, and 2) high access frequency. Both characteristics are true of PIM kernels (Section IV), meaning that FR-FCFS inherently favors PIM kernels. This is evident from Figure 8b, where MEM speedup contributes as little as 35% to the overall speedup (P5/VC1, P6/VC1), with the average contribution at 41% and 45% with VC1 and VC2 configurations, respectively. **FR-FCFS-Cap** solves the first unfairness issue by restricting the number of row buffer hits that bypass older requests, improving fairness and providing starvation freedom with the VC2 configuration. The policy still, however, suffers from the second source and can cause starvation with VC1 configuration (P3, P6). The CAP, set empirically to 32, also introduces more switches (Figure 10a), which reduces throughput by 3.7% and 2.8%, on average, under VC1 and VC2, respectively, compared to FR-FCFS.



(a) Average number of mode switches, normalized to FCFS (VC1).



(b) Additional MEM conflicts per switch.



(c) MEM drain latency (in DRAM cycles) per switch.

Fig. 10: Average number of mode switches (a) and MEM$\rightarrow$PIM switch overheads in terms of additional MEM conflicts (b) and the latency of draining the MEM queue (c).

BLISS builds upon FR-FCFS, but performs worse than it in both our key metrics. Depending on its blacklist schedule, BLISS devolves into either one of MEM-First, PIM-First, or FR-FCFS. Our analysis shows that it spends around 20%, 20%, and 60% time in each of those states with a blacklist threshold of 4. Performing a sweep of the blacklist threshold, we note that BLISS performs best with a lower threshold, indicating its tendency to converge toward FR-FCFS.

G&I, while designed to be a MEM-friendly policy [41], heavily favors PIM requests and even causes starvation with the VC2 configuration (Figure 8a, P5). Owing to PIM kernels' high request arrival rate, PIM requests cross the *high* threshold (set to 56) very quickly and fall below the *low* threshold (set to 32) only when either: 1) MEM requests create back pressure, or 2) the PIM kernel is nearing completion. VC2 mitigates interference at the interconnect, leading to starvation similar to MEM-First and PIM-First.

FR-RR-FCFS is the fairest policy in our characterization, achieving average fairness indices of 0.55 and 0.77 with VC1 and VC2 configurations, respectively. By switching mode on row buffer conflicts, FR-RR-FCFS ensures that all co-executing applications receive service and resolves the FCFS-inherent unfairness in FR-FCFS. However, the policy is sill prone to favoring applications with high locality since only row buffer conflicts switch the application being serviced. We see this in Figure 8b with P4/VC2, where the PIM kernel's (STREAM-Scale) high locality (99.6%) hurts GPU kernels' speedup and provides 60% of the throughput, on average.

B. Collaborative Co-execution

We next look at the collaborative scenario where all policies execute a decoder-only LLM (Section III). This scenario is different from the competitive scenario in that the primary metric is total execution time, not fairness. Figure 11 presents the speedup under each policy compared to sequential execution of the two kernels. Under VC1 configuration, all policies struggle to achieve any speedup. The key problem here is that QKV generation, running on GPU SMs, is the longer running of the two kernels, but the PIM kernels produce significantly more traffic and restrict the time MEM requests receive service. This allows a policy like G&I to work well. By favoring PIM requests, G&I is able to drain the interconnect and reduce congestion, allowing MEM requests to make progress. At the memory controller, MEM requests are issued once the MEM queue fills up and prevents PIM requests from coming in. While PIM-First should exhibit similar characteristics, it lags behind because of nearly double the number of switches and 6x higher switching latency.

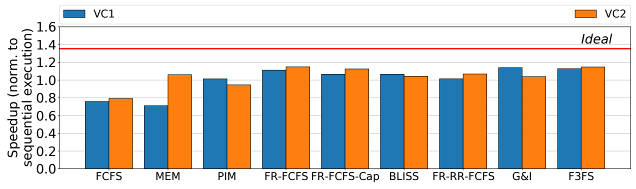


Fig. 11: LLM speedup for each policy with both a combined and separate VCs. The speedup is normalized to sequential execution of QKV generation and multi-head attention, while the *Ideal* represents the minimum of the two stages.

The results vary significantly in VC2 configuration. Since MEM and PIM requests do not interfere at the interconnect, PIM-favoring policies like G&I perform very close to sequential execution. MEM-First achieves a speedup >1 because it favors the slower running kernel, but it still limits parallelism. FR-FCFS shines here because it is able to maximize memory throughput, minimizing the overall execution time.

VII. First Mode-FR-FCFS (F3FS) - New and Improved PIM-Aware Memory Access Scheduling

We propose a new memory controller scheduling policy, called **First Mode-FR-FCFS** (shortened to **F3FS**), which attempts to improve both fairness and throughput. F3FS adds a new arbitration stage in front of FR-FCFS that favors requests in the current mode. That is, it implements the following priority order: 1) current mode first, 2) row buffer hit first, 3) oldest first. Within MEM and PIM modes, the queues utilize FR-FCFS and FCFS, respectively. By favoring requests in the current mode, F3FS improves throughput by maximizing locality and minimizing the switching frequency. To prevent one mode from starving another, F3FS also implements a *CAP* on the number of requests serviced in the current mode that bypass an older request in the other mode. Here, age is implemented as an incrementing ID assigned to each request as it enters the memory controller.

While fairness is an important metric for competitive co-execution that favors equal CAPs on MEM and PIM requests, collaborative co-execution may favor an unequal split of resources that results in an overall lower execution time. To support this, F3FS uses two CAPs, one each for MEM and PIM modes. In a collaborative scenario, the application can favor one type of kernel by setting a higher CAP value for it than the other. These asymmetric CAPs can also be configured by system software to enforce process priorities in competitive scenarios. We leave an exploration of the latter to future work.

A. Hardware Implementation

Figure 12 presents the architecture of the mode switch logic. In particular, the figure highlights the additions, deletions, and modifications for F3FS compared to the FR-FCFS switching policy. While F3FS introduces additional comparators and structures for counting the number of bypasses, it also gets rid of the per-bank conflict tracking that FR-FCFS performs. Such tracking goes beyond just maintaining a single bit for each bank and implementing the AND circuitry: for instance, the logic needs to track whether every bank has had at least one request issued before marking the next request as a conflict.

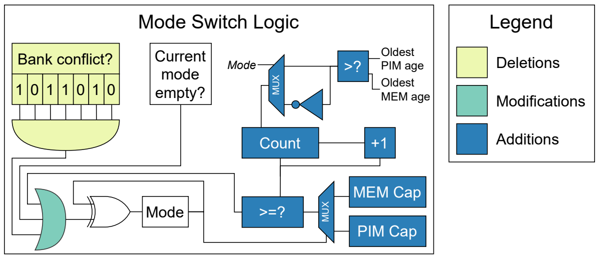


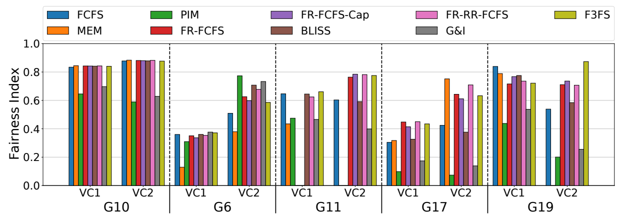
Fig. 12: Hardware overheads of F3FS in terms of the mode switch logic complexity, compared to FR-FCFS.

In order to quantify the area overheads of F3FS over FR-FCFS, we synthesized their mode switching logic on an AMD XCZU5EV FPGA [6] using Vitis HLS [5]. The synthesis reveals that F3FS requires 275/143 LUTs/flip-flops, compared to 377/88 for FR-FCFS.

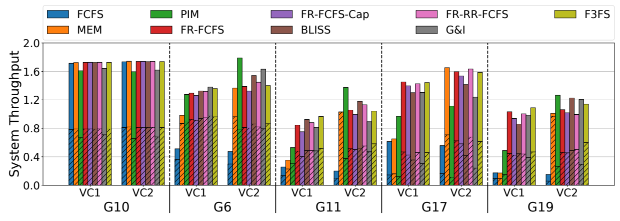
B. Evaluation

We compare F3FS with the baseline policies under both the VC1 and VC2 configurations. First, we look at competitive co-execution, where we use the same CAP (empirically set to 256) for both MEM and PIM to promote fairness. This CAP, determined from a sensitivity study, is strategically set to a multiple of the PIM RF size (eight per bank) to exploit the block structure of PIM kernels (Section II-B). Figure 8 shows the fairness and throughput improvements.

Competitive co-execution: F3FS performs the same or better than the best performing state-of-the-art policies in both VC1 and VC2 configurations. Under VC1 configuration, F3FS provides fairness comparable to FR-RR-FCFS, while achieving 1.8% and 5.1% higher average throughput compared to FR-FCFS and FR-RR-FCFS, respectively. This is a direct result of F3FS switching less frequently (Figure 10a) while paying comparable costs per switch (Figures 10b and 10c). Meanwhile, under VC2 configuration, F3FS outperforms FR-RR-FCFS in terms of both fairness and throughput by 4.7% and 2.6%, respectively, on average. This is a key result that highlights the throughput benefits of favoring current mode and fairness benefits of capping the wait time of each mode. Beyond averages, F3FS improves the worst-case fairness/throughput by 76.76%/28.98% under VC1 and by 146.22%/29.84% under VC2, respectively, compared to FR-RR-FCFS. Combining F3FS with our proposed interconnect changes yields average and best case fairness/throughput improvements of 48%/13% and 72%/22%, respectively, compared to a baseline single VC interconnect and FR-RR-FCFS policy. These improvements highlight how F3FS enhances the feasibility of concurrent host/PIM execution.



(a) Fairness Index (FI). FI of 0 indicates starvation.



(b) System Throughput. The shaded and non-shaded regions represent MEM and PIM speedups, respectively.

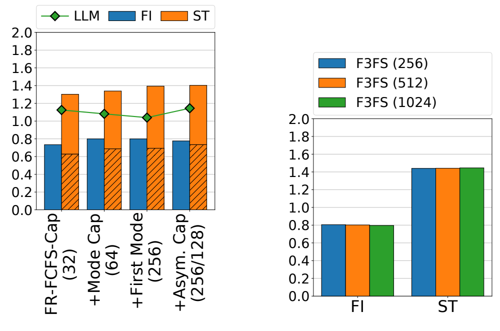
Fig. 13: Fairness (a) and throughput (b) of a compute intensive (G10) and four memory intensive (G6, G11, G17, G19) Rodinia kernels, averaged across all PIM kernels.

In order to evaluate how extremes in the memory intensity of the GPU kernels affects the performance of F3FS in a competitive scenario, Figure 13 shows the average fairness and throughput when a PIM kernel is executed with a compute intensive kernel (G10) or one of four of the most memory intensive Rodinia kernels (G6, G11, G17, G19), averaged across all PIM kernels (an orthogonal slice of Figure 8). The memory intensive kernels are picked based on our characterization in Figure 4. With the compute intensive kernel G10, there is very little variation in both fairness and throughput across scheduling policies and interconnect configurations, highlighting such applications' tolerance for memory access delays. Memory intensive kernels have more varied results. F3FS works well with G19, where interconnect traffic is high, but is filtered by the L2 cache, and is indicative of the common case of moderate memory traffic. F3FS is able to equalize queuing delays for MEM and PIM requests by using a symmetric CAP, while maintaining long enough phases to achieve BLP and RBHR comparable to standalone execution and minimize switching overheads. With G6 and G11, F3FS unfairly favors GPU kernels due to long MEM phases. G6 achieves higher BLP with F3FS than other policies, elongating the MEM drain latency per MEM→PIM switch. G6 also has long MEM phases because of its poor locality (average RBHR of 32%). F3FS, by equalizing the number of MEM and PIM requests served, inadvertantly leads to longer MEM phases because MEM requests take longer than PIM requests, on average. G11's high MEM request arrival rate ensures that MEM requests frequently execute up to the CAP. Even when the CAP is reached, the high MEM arrival rate often results in staying in MEM mode due the oldest request at the memory controller continuing to be MEM. G17's high RBHR results in smaller MEM phases, resulting in unfairly high PIM speedup due to the application's sensitivity to prolonged MEM queuing delays resulting from a PIM CAP of 256.

Collaborative co-execution: Figure 11 presents the speedup of the LLM under each policy. This is where F3FS's asymmetric CAP comes into play. We configure F3FS to use MEM/PIM CAPs of 256/128 and 64/64 in VC1 and VC2 configurations, respectively, based on a sensitivity study. Setting the asymmetric CAPs is a balancing act since throughput favors high CAPs while fairness favors lower ones. Starting with high values, the asymmetric CAP under VC1 configuration is lowered based on two principles: 1) a high enough PIM CAP to ensure consistent influx of MEM requests into the memory controller, and 2) a high enough MEM CAP to service as many MEM requests as possible without starving PIM requests. While this asymmetry helps in the VC1 configuration, VC2 configuration favors a symmetric CAP. To understand this, we discuss the MEM and PIM CAPs separately. For MEM CAP, increasing the value beyond 64 did not help since only 8% of MEM→PIM switches were triggered due to the CAP being exceeded. Meanwhile, for the PIM CAP, lowering the value below 64 had two implications: 1) increased switch overheads, and 2) fewer MEM requests in the MEM queue at the end of a PIM phase, reducing the memory controller's visibility into the GPU kernel's memory access stream and hurting locality. The chosen parameters allow F3FS to match the best performing policies in both the configurations (G&I in VC1 and FR-FCFS in VC2). Compared to FR-RR-FCFS, F3FS improves speedup by 11.23% and 7.37% in VC1 and VC2, respectively. These results highlight the flexibility of F3FS, showing how it can be dynamically configured to an application's needs.

C. Discussion

Ablation study: In order to better understand F3FS's performance improvements, we study the impact of its three components that differ from FR-FCFS-Cap: 1) CAP on the number of requests serviced in current mode (vs. row buffer hits), 2) prioritizing current mode first, and 3) the ability to use asymmetric CAPs on MEM and PIM modes. Figure 14a shows the incremental performance impact of the three components under VC2 configuration for P2 (averaged across all GPU kernels, except `kmeans` since it starves with FR-FCFS-Cap) and the LLM. The CAP for each stage is set separately to maximize competitive performance, and is listed in the figure. Moving the CAP from limiting row buffer hits to limiting requests in the current mode improves the average fairness index from 0.73 to 0.8 for P2, while reducing the LLM speedup by 4%. Next, favoring current mode brings about throughput improvements by reducing the number of switches, while still maintaining nearly the same fairness index. The LLM, on the other hand, drops to a speedup of 1.04. Finally, the last bar demonstrates the impact of asymmetric CAPs (MEM/PIM CAPs of 256/128 respectively to prioritize the slower MEM kernel). Asymmetry negatively impacts fairness in a competitive scenario, but benefits the LLM by reducing the queuing delay for MEM requests, improving speedup by 10% and pushing it higher than that of FR-FCFS-Cap.



(a) **(b)**

Fig. 14: (a) Impact of F3FS components on fairness index (FI) and system throughput (ST) of P2 and speedup of the LLM. The shaded and non-shaded ST regions represent MEM and PIM speedups, respectively. (b) Sensitivity of F3FS to interconnect queue size under VC2 configuration in terms of FI and ST across all GPU/PIM combinations.

Sensitivity to interconnect queue size: Figure 14b shows the performance sensitivity of F3FS to the interconnect queue size. The queue size is varied from half (256) to double (1024) the baseline size of 512. The figure shows how F3FS is largely agnostic to the queue size itself and neither benefits from longer nor is impeded by shorter queues.

VIII. Related Work

PIM architectures: A variety of PIM architectures have been proposed in the past decade, not all of which utilize separate PIM and MEM modes. Some architectures separate regular DRAM from PIM DRAM (e.g., UPMEM [19], [63]), while others place traditional processors near regular DRAM [3], [7], [9], [54]. Additionally, charge sharing techniques modify the memory controller to violate DRAM timing constraints and activate multiple rows within a subarray to perform bulk bitwise operations [18], [58], [59].

PIM integration: UM-PIM [70] is a hardware/software memory management scheme that optimizes accesses to PIM-enabled memory and traditional memory for locality and bandwidth, respectively. PIM-MMU [39] optimizes data transfers between PIM-enabled and traditional memory by using a dedicated Data Copy Engine that incorporates a PIM-aware memory scheduling policy. Like UM-PIM, PIM-MMU also uses separate optimized memory mapping schemes for each memory type. PyPIM [43] is an end-to-end programming framework for memristive PIM [14], [61] that incorporates a PIM ISA, a host driver, and a Python development library, along with a GPU-accelerated PIM simulator for testing and validation. PIM-Enabled Instructions (PEI) [2] is a locality-aware PIM offloading framework that utilizes ISA extensions to program an in-core PEI Computation Unit (PCU). The PCU executes the instructions either locally or on PIM based on the locality of its input operands. GraphPIM [49] is another PIM offloading framework designed for graph workloads. GraphPIM works by offloading atomic instructions that access an uncacheable PIM memory space to PIM units.

Memory controller scheduling: Memory request scheduling for CPUs in multi-application scenarios is a well-studied problem [36], [37], [46], [47], [62]. Most of these policies require thread-level state maintenance and communication between the host and the memory controller that, while tenable for CPUs, would be too expensive for GPUs. ITS and WEIS [32] are instruction throughput and weighted speedup optimizing policies that utilize LLC misses per kilo instructions and DRAM bandwidth, respectively, to prioritize applications. Both would devolve into MEM/PIM-First depending on their priority order. DASH [64] is a memory scheduler for accelerator-rich systems that provides quality-of-service to hardware accelerators executing real-time applications while ensuring CPU applications make progress whenever possible.

SMS [8] is a memory scheduling policy for shared DRAM CPU/GPU systems, where GPUs are often significantly more memory intensive than CPUs. While SMS works in the presented context, its batch granularity scheduling makes it unsuitable for concurrent host/PIM accesses. In particular, CPU/GPU batches that map to different banks can be serviced in parallel, but host/PIM batches can not. SMS does not take this exclusivity in account. G&I [41] is also a PIM-aware policy for bank-level PIM architectures. Our evaluation shows how the policy is PIM-biased and is outperformed by F3FS.

Host/PIM concurrency: LLMs can leverage host/PIM concurrency by overlapping QKV generation and MHA on host and PIM, respectively (AttAc! [53], NeuPIMs [24]) or by distributing fully connected layers between host and PIM at a head granularity (IANUS [57]). While AttAc! and IANUS take care to not submit MEM and PIM requests simultaneously, NeuPIMs proposes a dual row buffer architecture, one each for serving MEM and PIM requests. F3FS makes none of these assumptions and can be tuned based on application characteristics. Pimacolaba [25] proposes software and hardware optimization to parallelize Fast Fourier Transforms across GPU SMs and PIM FUs. Chopim [13] optimizes data layout and mapping to main memory, along with OS page coloring, to reduce PIM/host access interference and maximize main memory utilization. Pattnaik et al. [54] combine a GPU vs. PIM affinity model with a dynamic execution time prediction model to dispatch GPU kernels to GPU cores or PIM FUs, with the goal of minimizing overall execution time. Most application scheduling and memory management schemes can be combined with our proposals to improve system utilization.

IX. Summary

Integrating PIM-enabled main memories into existing systems remains an open challenge. In this paper, we show how the sharing of the interconnect and main memory controller by *MEM*ory and *PIM* requests can severely degrade application-level fairness and system-level throughput. The memory intense nature of PIM kernels can overwhelm the memory controller under contention and create back pressure in the interconnect, hurting any co-executing application's memory performance. Our characterization of a GPU/PIM system shows that such contention causes MEM request arrival rate at the memory controller to drop by up to 95%. We propose a two-step solution to remedy this. First, we modify the interconnect and add a separate virtual channel (VC) for PIM requests to mitigate MEM/PIM interference. Second, we introduce a new memory controller scheduling policy, called F3FS, that improves: 1) fairness, by providing equal service to the two request types, and 2) throughput, by minimizing MEM/PIM mode switching frequency. When evaluated on 180 competing GPU/PIM kernel combinations, our solution achieved up to 72% and 22% better fairness and throughput, respectively, compared to FR-RR-FCFS policy with a single VC. F3FS is also tunable at runtime and can be configured to favor one request type over another, allowing a GPT-3 like LLM to execute 13.14% faster when compared to the same baseline. Beyond performance averages, F3FS also improves worst-case throughput and fairness metrics, enhancing the feasibility of GPU/PIM co-execution.

Acknowledgements

This work was supported in part by U.S. National Science Foundation grant CNS-1900803. We would like to thank the anonymous program committee and artifact evaluation committee reviewers for their feedback.

References

[1] S. Aga, N. Jayasena, and M. Ignatowski, "Co-ML: A Case for Collaborative ML Acceleration Using near-Data Processing," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 506–517, event-place: Washington, District of Columbia, USA. [Online]. Available: https://doi.org/10.1145/3357526.3357532

[2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 336–348, event-place: Portland, Oregon. [Online]. Available: https://doi.org/10.1145/2749469.2750385

[3] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O'Halloran, D. Chen, J. Xiong, D. Kim, W.-m. Hwu, and N. S. Kim, "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.

[4] AMD, "AMD ROCm HIP Documentation: Stream Management." [Online]. Available: https://rocm.docs.amd.com/projects/HIP/en/latest/doxygen/html/group___stream.html

[5] AMD, "AMD Vitis Unified Software Platform." [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis.html

[6] AMD, "AMD Zynq UltraScale+ MPSoCs." [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-ultrascale-plus-mpsoc.html

[7] M. Asri, C. Dunham, R. Rusitoru, A. Gerstlauer, and J. Beard, "The Non-Uniform Compute Device (NUCD) Architecture for Lightweight Accelerator Offload," in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2020, pp. 38–45.

[8] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012, pp. 416–427.

[9] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "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*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 316–331, event-place: Williamsburg, VA, USA. [Online]. Available: https://doi.org/10.1145/3173162.3173177

[10] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," 2020, arXiv: 2005.14165.

[11] S. Charagulla, "High-Bandwidth Memory Ready for AI Prime Time: HBM2e vs. GDDR6," May 2019. [Online]. Available: https://www.eeweb.com/high-bandwidth-memory-ready-for-ai-prime-time-hbm2e-vs-gddr6/

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[13] B. Y. Cho, Y. Kwon, S. Lym, and M. Erez, "Near Data Acceleration with Concurrent Host Access," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 818–831.

[14] L. Chua, "Memristor-The missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, 1971.

[15] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[16] 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.

[17] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: Association for Computing Machinery, 2012, pp. 37–48, event-place: London, England, UK. [Online]. Available: https://doi.org/10.1145/2150976.2150982

[18] 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*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, pp. 100–113, event-place: Columbus, OH, USA. [Online]. Available: https://doi.org/10.1145/3352460.3358260

[19] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System," *IEEE Access*, vol. 10, pp. 52 565–52 608, 2022.

[20] R. Hadidi, B. Asgari, B. A. Mudassar, S. Mukhopadhyay, S. Yalamanchili, and H. Kim, "Demystifying the characteristics of 3D-stacked memories: A case study for Hybrid Memory Cube," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, 2017, pp. 66–75.

[21] M. Harris, "GPU Pro Tip: CUDA 7 Streams Simplify Concurrency," Jan. 2015. [Online]. Available: https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

[22] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. N. Vijaykumar, "Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 372–385.

[23] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[24] G. Heo, S. Lee, J. Cho, H. Choi, S. Lee, H. Ham, G. Kim, D. Mahajan, and J. Park, "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, Volume 3*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 722–737, event-place: La Jolla, CA, USA. [Online]. Available: https://doi.org/10.1145/3620666.3651380

[25] M. A. Ibrahim and S. Aga, "Pimacolaba: Collaborative Acceleration for FFT on Commercial Processing-In-Memory Architectures," in *Proceedings of the 2024 International Symposium on Memory Systems*, ser. MEMSYS '24. New York, NY, USA: Association for Computing Machinery, 2024, event-place: Washington DC, DC, USA.

[26] I. R. Ivanov, O. Zinenko, J. Domke, T. Endo, and W. S. Moses, "Retargeting and Respecializing GPU Workloads for Performance Portability," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2024, pp. 119–132. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CGO57630.2024.10444828

[27] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *2012 Symposium on VLSI Technology (VLSIT)*, 2012, pp. 87–88.

[28] JEDEC, "High Bandwidth Memory (HBM) DRAM (JESD235D)," Mar. 2021.

[29] JEDEC, "Graphics Double Data Rate (GDDR6) SGRAM Standard (JESD250D)," May 2023.

[30] Z. Jin, "The Rodinia Benchmark Suite in SYCL," Jun. 2020. [Online]. Available: https://www.osti.gov/biblio/1631460

[31] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-7. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1–8, event-place: Salt Lake City, UT, USA. [Online]. Available: https://doi.org/10.1145/2588768.2576780

[32] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of GPU

Memory System for Multi-Application Execution," in *Proceedings of the 2015 International Symposium on Memory Systems*, ser. MEMSYS '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 223–234, event-place: Washington DC, DC, USA. [Online]. Available: https://doi.org/10.1145/2818950.2818979

[33] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim, "HBM (High Bandwidth Memory) DRAM Technology and Architecture," in *2017 IEEE International Memory Workshop (IMW)*, 2017, pp. 1–4.

[34] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[35] J. S. Kim, D. Senol Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies," *BMC Genomics*, vol. 19, no. 2, p. 89, May 2018. [Online]. Available: https://doi.org/10.1186/s12864-018-4460-0

[36] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[37] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 65–76.

[38] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim, Y. Cho, J. G. Kim, J. Choi, H.-S. Shin, J. Kim, B. Phuah, H. Kim, M. J. Song, A. Choi, D. Kim, S. Kim, E.-B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. Song, J. Youn, K. Sohn, and N. S. Kim, "25.4 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.

[39] D. Lee, B. Hyun, T. Kim, and M. Rhu, "PIM-MMU: A Memory Management Unit for Accelerating Data Transfers in Commercial PIM Systems," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 627–642.

[40] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, K. Vladimir, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, J. Lee, D. Ko, Y. Jun, K. Cho, I. Kim, C. Song, C. Jeong, D. Kwon, J. Jang, I. Park, J. Chun, and J. Cho, "A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications," in *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 65, 2022, pp. 1–3.

[41] S. Lee, S. Lee, M. Seo, C. Park, H.-J. Lee, W. Shin, and H. Kim, "A High-Performance Scheduling Algorithm for Mode Transition in PIM," in *2021 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, 2021, pp. 1–4.

[42] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 43–56.

[43] O. Leitersdorf, R. Ronen, and S. Kvatinsky, "PyPIM: Integrating Digital Processing-in-Memory from Microarchitectural Design to Python Tensors," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 1632–1647.

[44] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 188–201, 1999.

[45] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A Modern Primer on Processing in Memory," 2022, arXiv: 2012.03112.

[46] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 146–160.

[47] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," in *2008 International Symposium on Computer Architecture*, 2008, pp. 63–74.

[48] A. Nag and R. Balasubramonian, "OrderLight: Lightweight Memory-Ordering Primitive for Efficient Fine-Grained PIM Computations," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 298–310, event-place: Virtual Event, Greece. [Online]. Available: https://doi.org/10.1145/3466752.3480103

[49] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 457–468.

[50] NVIDIA, "CUDA Toolkit Documentation: Stream Management," Apr. 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html

[51] NVIDIA, "Multi-Instance GPU," Aug. 2024. [Online]. Available: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/

[52] NVIDIA, "Multi-Process Service," Jun. 2024. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[53] J. Park, J. Choi, K. Kyung, M. J. Kim, Y. Kwon, N. S. Kim, and J. H. Ahn, "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, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 103–119, event-place: La Jolla, CA, USA. [Online]. Available: https://doi.org/10.1145/3620665.3640422

[54] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling techniques for GPU architectures with processing-in-memory capabilities," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016, pp. 31–44.

[55] B. R. Rau, "Pseudo-randomly interleaved memory," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ser. ISCA '91. New York, NY, USA: Association for Computing Machinery, 1991, pp. 74–83, event-place: Toronto, Ontario, Canada. [Online]. Available: https://doi.org/10.1145/115952.115961

[56] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 128–138, event-place: Vancouver, British Columbia, Canada. [Online]. Available: https://doi.org/10.1145/339647.339668

[57] M. Seo, X. T. Nguyen, S. J. Hwang, Y. Kwon, G. Kim, C. Park, I. Kim, J. Park, J. Kim, W. Shin, J. Won, H. Choi, K. Kim, D. Kwon, C. Jeong, S. Lee, Y. Choi, W. Byun, S. Baek, H.-J. Lee, and J. Kim, "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, Volume 3*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 545–560, event-place: La Jolla, CA, USA. [Online]. Available: https://doi.org/10.1145/3620666.3651324

[58] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 185–197.

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

[60] H. Shin, D. Kim, E. Park, S. Park, Y. Park, and S. Yoo, "McDRAM: Low Latency and Energy-Efficient Matrix Computations in DRAM," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2613–2622, 2018.

[61] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, May 2008. [Online]. Available: https://doi.org/10.1038/nature06932

[62] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 3071–3087, 2016.

[63] UPMEM, "UPMEM PIM Technical paper," Aug. 2022. [Online]. Available: https://www.upmem.com/technology/

[64] H. Usui, L. Subramanian, K. K.-W. Chang, and O. Mutlu, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, Jan. 2016, place: New York, NY, USA Publisher: Association for Computing Machinery. [Online]. Available: https://doi.org/10.1145/2847255

[65] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: A big data benchmark suite from internet services," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 488–499.

[66] L. Wu, R. Sharifi, M. Lenjani, K. Skadron, and A. Venkat, "Sieve: Scalable In-situ DRAM-based Accelerator Designs for Massively Parallel k-mer Matching," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 251–264.

[67] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995, place: New York, NY, USA Publisher: Association for Computing Machinery. [Online]. Available: https://doi.org/10.1145/216585.216588

[68] C. Xie, S. L. Song, J. Wang, W. Zhang, and X. Fu, "Processing-in-Memory Enabled Graphics Processors for 3D Rendering," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 637–648.

[69] Y. J. Yoon, N. Concer, M. Petracca, and L. P. Carloni, "Virtual Channels and Multiple Physical Networks: Two Alternatives to Improve NoC Performance," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 12, pp. 1906–1919, 2013.

[70] Y. Zhao, M. Gao, F. Liu, Y. Hu, Z. Wang, H. Lin, J. Li, H. Xian, H. Dong, T. Yang, N. Jing, X. Liang, and L. Jiang, "UM-PIM: DRAM-based PIM with Uniform & Shared Memory Space," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 644–659.

Appendix

A. Abstract

This artifact appendix describes how to reproduce key results from the paper. The artifact includes a modified version of the GPGPU-Sim 4.0.1 simulator, the PIM benchmark suite, and the Rodinia benchmark suite with input data. The modified GPGPU-Sim implements the baseline and proposed interconnect architecture and memory controller scheduling policies, along with modifications to some CUDA APIs to provide more control over concurrent kernel launches. The artifact can be set up easily using the provided Dockerfile.

B. Artifact check-list (meta-information)

- **Algorithm:** F3FS, a memory controller scheduling policy.
- **Program:** GPGPU-Sim, Rodinia benchmark suite, PIM benchmarks.
- **Data set:** Modified Rodinia benchmark suite inputs.
- **Hardware:** Dual core x86-64 based CPU and 8GB memory.
- **Metrics:** Fairness index, system throughput, number of mode switches, conflicts per switch, drain latency per switch, LLM speedup.
- **Output:** GPGPU-Sim output statistics and plots generated using matplotlib.
- **How much disk space required (approximately)?:** 20 GB.
- **How much time is needed to prepare workflow (approximately)?:** 15 minutes.
- **How much time is needed to complete experiments (approximately)?:** 2 weeks.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Creative Commons 4.0
- **Data licenses (if publicly available)?:** Creative Commons 4.0
- **Workflow automation framework used?:** Docker.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.15164086

C. Description

1) How to access: The artifact is available on both GitHub and Zenodo at the following links. We recommend using the Dockerfile to set up the artifact.

- Zenodo: https://zenodo.org/records/15164086
- GitHub:
  - Dockerfile: https://gist.github.com/Sacusa/47801d133eda38317bc8fc84013ed041
  - Simulator: https://github.com/Sacusa/GPGPU-Sim-4.0.1/tree/ISPASS_2025
  - Benchmarks: https://github.com/Sacusa/PIM_apps/tree/ISPASS_2025

2) Hardware dependencies: Recent x86-64 based CPU with at least 2 cores and 8GB memory. There are a total of 3258 simulations, so more cores and memory would help.

3) Software dependencies: This artifact only requires Docker to run.

4) Data sets: Input data is generated using scripts that come with the Rodinia benchmark suite.

D. Installation

The steps below detail how to build a Docker image and start a container for the artifact. This assumes that the user already has Docker installed.

1) Download the Dockerfile from either GitHub or Zenodo.
2) Navigate to the downloaded file from a terminal and execute the following :
```
docker build -t ispass2025:latest .
```
3) Once the build is complete, start a container by running the following command:
```
docker run -i -t --name <name> ispass2025:latest
```
Replace <*name*> with a name for the container. This command will create a new container and attach to its terminal.
4) To start the container again in the future, run:
```
docker start <name>
```
And connect to it by running:
```
docker attach <name>
```

E. Experiment workflow

Navigate to:
```
/opt/PIM_apps/STREAM
```
and launch the script:
```
run_baseline.sh 8
```
to run the baseline PIM experiments. This will simulate PIM kernels running alone. Next, navigate to:
```
/opt/PIM_apps/rodinia_3.1_pim/cuda
```
and run the script:
```
launch_ispass2025.sh
```
to run the baseline Rodinia and LLM experiments, followed by the competitive and collaborative experiments. While the

results presented in the paper require a total of 3258 simulations, the script can be modified to run a subset of experiments for faster reproduction of results.

F. Evaluation and expected results

The directory:
`/opt/PIM_apps/rodinia_3.1_pim/cuda/scripts`
contains plotting scripts to visualize key results from our paper. Once the simulations finish, the following figures from the paper can be regenerated using the respective scripts in the directory:

- Figure 6: `plot_mem_arrival_rate.py`
- Figure 8a: `plot_fairness_index.py`
- Figure 8b: `plot_throughput.py`
- Figure 10a: `plot_num_switches.py`
- Figures 10b, 10c: `plot_switch_overheads.py`
- Figure 11: `plot_llm_speedup.py`

G. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae