OrderLight: Lightweight Memory-Ordering Primitive for Efficient Fine-Grained PIM Computations

Anirban Nag
anirban@it.uu.se
Uppsala University
Uppsala, Sweden

Rajeev Balasubramonian
rajeev@cs.utah.edu
University of Utah
Salt Lake City, USA

ABSTRACT

Modern workloads such as neural networks, genomic analysis, and data analytics exhibit significant data-intensive phases (low compute to byte ratio) and, as such, stand to gain considerably by using processing-in-memory (PIM) solutions along with more traditional accelerators. While PIM has been researched extensively, the granularity of computation offload to PIM and the granularity of memory access arbitration between host and PIM, as well as their implications, have received relatively little attention. In this work, we first introduce a taxonomy to study the design space whilst considering these two aspects. Based on this taxonomy, we observe that much of PIM research to date has largely relied on coarse-grained approaches which, we argue, have steep costs (incompatibility with mainstream memory interfaces, prohibition of concurrent host accesses, and more). To this end, we believe that better support for fine-grained approaches is warranted in accelerators coupled with PIM-enabled memories.

A key challenge in the adoption of fine-grained PIM approaches is enforcing memory ordering. We discuss how existing memory ordering primitives (fences) are not only insufficient but their large overheads render them impractical to support fine-grain computation offloads and arbitration. To address this challenge, we make the key observation that the core-centric nature of memory ordering is unnecessary for PIM computations. We propose a novel lightweight memory ordering primitive for PIM use cases, *OrderLight*, which moves away from core-centric ordering enforcement and considerably reduces the overheads of enforcing correctness. For a suite of key computations from machine learning, data analytics, and genomics, we demonstrate that *OrderLight* delivers 5.5× to 8.5× speedup over traditional fences.

CCS CONCEPTS

• **Computer systems organization** → *Special purpose systems.*

KEYWORDS

Processing-in-Memory, PIM Taxonomy, Fine-grain Offload, Fine-grain Arbitration, Memory-centric Ordering

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480103

ACM Reference Format:
Anirban Nag and Rajeev Balasubramonian. 2021. OrderLight: Lightweight Memory-Ordering Primitive for Efficient Fine-Grained PIM Computations. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480103

1 INTRODUCTION

Many critical workloads today include significant data-intensive phases (few computations per byte of data) along with compute-intensive portions. For example, convolutional neural networks are comprised of both compute-intensive convolutions and data-intensive computations such as batch normalization [1], feature map addition [1], and classifier layers [49]. Similarly, genomic analysis consists of both compute-intensive (string matching) and data-intensive (sequence filtering [48]) portions. While general-purpose or domain specific accelerators such as GPU and TPU [24] are able to tackle the compute-intensive portions of applications, data-intensive phases remain limited by memory bandwidth.

Processing-in-memory (PIM)[1] solutions move compute close to memory arrays, providing significant memory bandwidth advantage over host[2] processors. As such, PIM solutions are uniquely positioned to speedup data-intensive portions of modern workloads. Given modern workloads comprise both data-intensive and compute-intensive phases as discussed above, it is increasingly important to couple host accelerators with memory equipped with compute capability (i.e., PIM).

While many PIM designs have been proposed, two aspects that have received limited attention are: 1) the granularity of operations offloaded to PIM, and 2) the interaction between PIM computations and host memory accesses (i.e., whether host and PIM are allowed concurrent access to memory). We believe that carefully considering these aspects and their tradeoffs helps to better serve the needs of modern workloads. To this end, we first develop a taxonomy of PIM designs based on the granularity of offloaded PIM computations and the granularity of arbitration with host memory accesses. Further, to consider both of these aspects in tandem, we focus on *temporal* (time) granularity as opposed to data granularity.

Based on our taxonomy, we observe that PIM designs spanning the past several decades [2, 7, 9, 11–13, 15–17, 20, 21, 25, 26, 28, 29, 35, 42, 43, 47] have mostly utilized coarse-grained offload where the host ships entire computations to memory-side logic as depicted in Figure 1. Although such coarse-grain offload provides simplicity of design, we argue that it has steep costs. First, such

[1] In this work, we use *PIM* to refer to both in-memory-array and near-memory-array logic as the proposed work is largely agnostic to this distinction.
[2] We use the term *host* to refer to any processor or accelerator attached, but external, to PIM-enabled memory module(s).

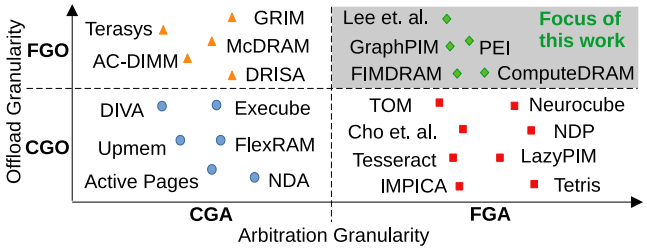

Figure 1: Taxonomy of PIM based on *temporal* coarse-grained and fine-grained offload (CGO & FGO) and coarse-grained and fine-grained arbitration (CGA & FGA) for host memory accesses with examples from the literature.

an approach requires more complex memory-side logic to orchestrate offloaded computations (e.g., instruction/operation sequencing within the PIM devices). Second, assuming concurrent memory accesses from the host, it requires moving the memory controller to the memory module, along with arbitration between PIM and host accesses [38]. This is often impractical in commodity systems. Alternately, prohibiting concurrent memory accesses by host and PIM computations can lead to drastic reductions in utilization of the host processor when PIM is heavily used.

Based on the taxonomy, we also identify an emerging class of PIM designs [3, 14, 32, 34, 39] that perform fine-grained offload, in a temporal sense, that overcome several shortcomings of the coarse-grained approach discussed above. First, it keeps the host processor in charge of orchestrating PIM computation, thus, greatly simplifying memory-side logic to primarily cater to data-intensive computations which are most suitable for PIM. Second, it keeps the memory interface compatible with the fundamentals of existing memory interfaces even in the presence of PIM. Finally, it allows the memory controller to schedule PIM commands interleaved with normal loads/stores, allowing concurrent operation of the host and PIM.

While the fine-grained approach for PIM is advantageous in many respects, it does have concomitant challenges. A key challenge that we focus on in this work is the *ordering* needed amongst the fine-grained PIM commands. Existing memory ordering primitives (aka fences) that are used by host computations to enforce ordering of memory operations have severe overheads [30] and are often used sparingly by programmers. As such, they are ill-equipped to efficiently enforce ordering for fine-grained PIM commands.

To address these challenges, we make the key observation in this work that, unlike existing memory ordering primitives which are *core-centric*, the ordering requirement for PIM commands is *memory-centric*. Consider a PIM computation where two arrays are added and stored in a third array. This computation is accomplished by a sequence of fine-grained PIM commands that read the two inputs, add them, and write the result back to memory, all of which need to be ordered with respect to one another for correctness. Unlike core-centric ordering primitives where values have to be read at the core before the write can be completed (again at the core), the ordering requirement for PIM commands has to be enforced at the PIM unit performing the computation (i.e., at the memory).

We exploit this observation by proposing *OrderLight*, a lightweight memory ordering primitive where ordering is enforced at the memory controller instead of the core by sending *OrderLight* packets along with PIM instructions to the memory controller. In order to orchestrate PIM computations, the core simply issues a series of PIM instructions and *OrderLight* primitives down the memory pipe, without stalling. We demonstrate the performance benefits of moving away from a core-centric enforcement of memory ordering using a suite of computations from relevant applications (machine learning, genomics, data analytics) and streaming benchmarks. Overall, we make the following contributions:

- We introduce a taxonomy of PIM designs based on the temporal granularity of both offloaded PIM computations and arbitration with host memory accesses and use it to discuss characteristics desirable in a PIM design in today's computing landscape.
- We make the observation that existing memory ordering primitives are ill-equipped to support fine-grained PIM designs (a key sub-class within the above taxonomy) as existing ordering primitives are *core-centric*.
- To address this challenge, we propose a new lightweight memory ordering primitive, *OrderLight*, which moves away from core-centric ordering enforcement and considerably reduces the overheads of enforcing correctness for PIM.
- For a suite of applications (machine learning, data analytics, genomics) and streaming benchmarks, we demonstrate that *OrderLight* delivers 5.5× to 8.5× improvement over existing memory ordering primitives.

2 BACKGROUND

We first discuss relevant workloads and GPUs as potential host accelerators that can be coupled with PIM.

2.1 Data-intensive Phases in Modern Workloads

Increasing data-centric processing has led to many modern workloads having both data-intensive (few computations per byte of data) and compute-intensive phases. While general purpose and specialized accelerators can tackle compute-intensive portions, PIM solutions stand best to speedup data-intensive portions. We discuss some of these workloads below.

Machine Learning: Workloads such as neural network (DNN, RNN, etc.) training and inference consist of multiple layers of computation. Layers such a convolutions are often formulated as matrix-matrix multiplication operations and are compute-intensive. Other layers such as feature-map addition (e.g., in residual networks [18]), batch normalization [22], and fully-connected (during inference) have low compute-to-byte ratios and are data-intensive [1]. Feature-map addition sums neuron activations of two layers (two vectors) to feed as input to a third layer (output vector). Batch normalization scales and biases an input vector of neuron activations. Fully-connected layers in inference perform a series of dot product operations of a large input activation vector with a large number of weight vectors. Profiling shows that data-intensive computations constitute approximately 32% of the training runtime of ResNet50 on current GPU hardware [1].

Genomic Analysis: Sequence alignment is a key step in genomic analysis and is performed by aligning small sequences (called reads) against a reference genome. A read is aligned against the reference at a set of candidate locations using a dynamic programming step (compute-intensive). To reduce the set of candidate locations, a filtering algorithm is used. The filtering algorithm computes simpler operations such as Hamming distance or dot-product at a large number of candidate locations and is data-intensive. Filtering has been shown to constitute 65% of sequence alignment runtime [28].

Data Analytics: Data analytics for unstructured data, such as unlabeled text or images, typically requires two main steps: feature extraction and clustering. Feature extraction is often performed using neural networks that provide a feature vector for a word, a sentence, or an image (compute-intensive convolutions). The feature vectors obtained from this first step are then used to cluster the data points within a large dataset using algorithms such as Kmeans and Histogram. Kmeans and Histogram are data-intensive as they require sifting through large amount of data with simple computations (Kmeans: distance from center, Histogram: bin update).

2.2 Host Accelerator - GPU

Several of the modern workloads discussed in Section 2.1 benefit from having a GPU as the host accelerator (e.g., GPU is preferred for ML training [4]). As such, the system we evaluate assumes a GPU coupled with PIM-enabled (Section 4.1) High Bandwidth Memory (HBM) [23] as our baseline (we discuss further reasons for this baseline in Section 4.3). However, the ideas and architectural innovations discussed in the paper are applicable broadly across other forms of hosts and PIM organizations as well.

A GPU consists of multiple cores, known as Streaming Multiprocessors (SMs) or Compute Units (CUs) in NVIDIA and AMD terminology, respectively. Each SM has one or more SIMD units to issue instructions from a vector of threads called warps or wavefronts. The GPU SMs issue memory requests through their load/store (LDST) units. Each SM has a private L1 cache, a texture cache, a constant cache, a software managed scratchpad, and a shared instruction cache. The SMs are connected to a shared L2 cache via an interconnection network. Each memory channel is associated with its own L2 slice. There are multiple memory controllers, one per channel. Physical memory is interleaved at chunk granularity (e.g., 256B chunks) across memory channels.

3 TAXONOMY OF PIM OFFLOAD AND ARBITRATION

In this section, we present our taxonomy for PIM designs with a focus on the temporal granularity of two aspects: (i) offloaded PIM computations, and (ii) arbitration of PIM computation and host memory accesses. We also use this taxonomy to discuss characteristics desirable in a PIM design in today's computing landscape. Note that "temporal granularity" refers to the amount of time consumed by an offloaded PIM computation, and not the amount of data it operates on[3]. Further note that, as discussed in Section 2, given the heterogeneous nature of modern workloads, we consider PIM designs coupled with host accelerators which tackle both compute and data-intensive phases. We do not consider designs where memory is the main compute unit [35, 46] (i.e., no host processor).

[3]While there is some correlation between the temporal granularity of a task and the amount of data accessed, we note that the relationship is not uniform across PIM architectures. For example, a temporally fine-grain bitwise operation on an entire DRAM row may touch several KB of data but still complete within a single row operation's worth of time.

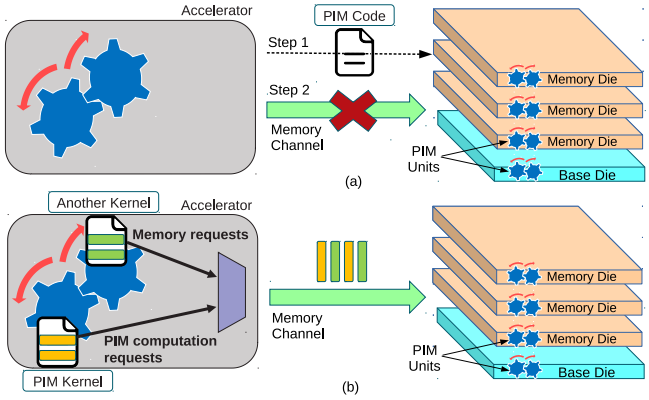

Figure 2: (a) Coarse-grained arbitration disallows concurrent host and PIM memory accesses, (b) Fine-grained arbitration interleaving of PIM and host memory accesses.

3.1 Coarse-grain Offload and Fine-grain Arbitration

We term designs which ship entire PIM computations to memory-side logic but allow host and PIM computation to arbitrate for memory accesses at fine granularity (typically, individual load/stores) as coarse-grain offload and fine-grain arbitration (CGO/FGA) designs [2, 7, 9, 15, 20, 21, 26, 43]. Such designs require complex memory-side logic to orchestrate PIM computation. Further, they enable fine-grain arbitration between PIM and host accesses by moving memory scheduling from the host to the memory module using transactional host memory interfaces, often based on the Hybrid Memory Cube [38]. We note that none of the currently available mainstream memory interfaces provide the transactional semantics required by these designs [44].

3.2 Coarse-grain Offload and Coarse-grain Arbitration

Coarse-grain offload and coarse-grain arbitration (CGO/CGA) designs follow offload mechanisms similar to CGO/FGA designs above, but they disallow concurrent memory accesses from host and PIM computations [11–13, 25, 29, 42]. As a result, these designs (depicted in Figure 2a) render system memory inaccessible to the host during PIM computations. This is undesirable in datacenters (and other multi-use environments) as it can impact QoS guarantees of all tasks scheduled on the host and adversely affect the achievable utilization of datacenter resources. Further, this approach also places a lower bound on the minimum computation granularity for PIM offloads as they must be large enough to justify draining the host's memory pipeline prior to launching the PIM computation and refilling it after completion of PIM execution.

3.3 Fine-grain Offload and Coarse-grain Arbitration

PIM designs with fine-grain offload and coarse-grain arbitration (FGO/CGA) offload PIM computations at fine granularity (typically temporally equivalent to individual loads/stores) which simplify the memory-side logic to only support data-intensive computations and not associated orchestration logic [16, 17, 28, 35, 47]. However, such designs suffer from the drawbacks of coarse-grain arbitration as outlined in Section 3.2.

3.4 Fine-grain Offload and Fine-grain Arbitration

PIM designs with fine-grain offload and fine-grain arbitration (FGO/FGA) keep memory-side logic devoid of PIM orchestration overheads and, at the same time, allow fine-grain arbitration of host and PIM memory accesses (depicted in Figure 2b). While historically scarce, a few recent research efforts fall into this class [3, 14, 32, 34, 39]. Furthermore, this approach shows good versatility, as the designs utilizing this approach span transactional memory interfaces [3] as well as very minor variations of mainstream JEDEC memory standards [32, 34].

3.5 Desirable PIM Characteristics

With the above taxonomy in mind, in this work, we observe that there is significant value in FGO/FGA PIM designs as they truly enable host and PIM computations to execute concurrently which is often highly beneficial for modern workloads. Further, they reduce memory-side logic complexity (no orchestration needed within memory modules) and also broaden PIM usage by allowing even small segments of computation to be effectively offloaded to PIM. In addition, FGO/FGA designs are compatible with mainstream memory interfaces such as DDR, HBM, GDDR, and LPDDR. As such, we believe that improving the efficiency of such designs is warranted.

4 CHALLENGE: ORDERING OF PIM INSTRUCTIONS

While in Section 3 we discussed the desirability of FGO/FGA PIM designs, we discuss in this section a key challenge associated with these designs, which is the focus of this work. To appropriately highlight this challenge, we first discuss a generic and parameterized PIM unit, followed by an example set of fine-grained PIM operations the PIM unit performs. Finally, we use both of the above to discuss how existing memory ordering primitives fall short of efficiently supporting ordering requirements of fine-grained PIM instructions needed in FGO/FGA designs, thus motivating the need for an efficient memory ordering primitive geared towards this use case.

4.1 Generic and Parameterized PIM Compute Unit

Our proposed work is agnostic to the specific memory-side logic and is applicable whenever FGO/FGA PIM designs are employed (more details in Section 4.3). Consequently, in this work, we consider a generic PIM compute unit which processes fine-grained PIM commands as depicted in Figure 3. Further, we parameterize this unit to study a variety of PIM solutions. The PIM unit we consider consists of a SIMD ALU coupled with temporary storage (labelled TS in Figure 3). The SIMD nature of the ALU allows effective utilization of the high bandwidth typically available to PIM designs, and the temporary storage buffers operands read from memory or results to be stored to memory. Note that while Figure 3 shows a PIM unit that may reside on a 3D-stacked die separate from the memory arrays, alternatively, the PIM unit may be placed close to the arrays (e.g., near a memory bank or a memory sub-array) representing a broad swath of different PIM solutions. Further, specializations and different cardinalities of the PIM unit can also be considered. Depending on the placement and number of units, different bandwidth multiplication factors over host-available memory bandwidth is realized collectively by the PIM units. In our evaluations, we sweep this bandwidth multiplier to study disparate PIM solutions. Further, we also study the efficacy of our ideas by varying the size of temporary storage associated with these PIM units.

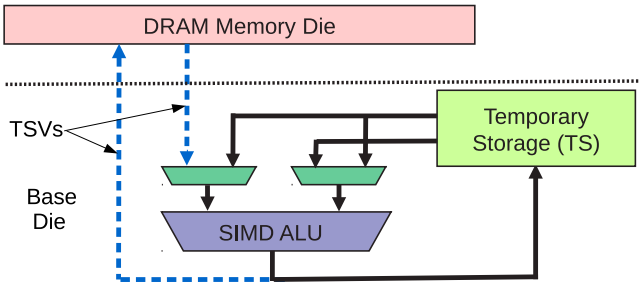

Figure 3: A generic and and parameterized PIM compute unit with temporary storage (TS) and SIMD ALU. One or more such compute units may be placed in multiple locations per channel to obtain bandwidth multiplication over host.

4.2 Fine-grained PIM commands

We discuss in this section the nature of fine-grained PIM commands that we consider in this work. As with any memory access to DRAM, fine grained PIM commands incur *precharge* and *row activate* operations on operand accesses from memory. Further, we assume data movement operations similar to loads and stores are used to move data from/to an activated DRAM row to temporary storage associated with PIM and RISC-like instructions are used to orchestrate PIM computations.

Consider a simple PIM computation where two vectors (a and b) are added and stored in a third vector (c): $c[i] = a[i] + b[i]$. We envision the orchestration of such a computation using our generic PIM compute unit from Section 4.1 by using a sequence of fine-grained PIM commands as shown in Figure 4. Specifically, it will first load the input values into temporary storage. However, unlike normal host loads, these commands achieve higher bandwidth to the memory associated with the PIM unit (line 2). This is followed by PIM computation commands (line 5) to add the input operands together and store the result back to temporary storage. Finally, the result is stored to memory ($vector\ c$) (line 7).

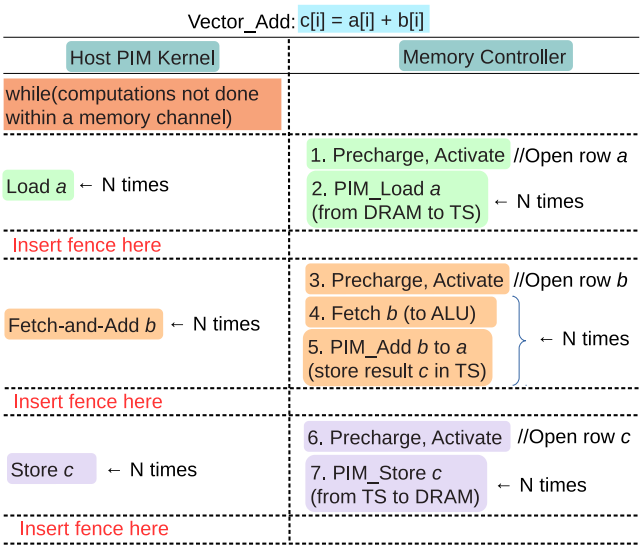

Figure 4: Host PIM kernel and fine-grained commands sent by the memory controller for the vector_add kernel. The computation is tiled to handle N operands at a time, such that the temporary storage size is not exceeded.

Note that based on available temporary storage size, multiple chunks of input operands can be saved (N commands, line 2) before adding them (N back-to-back compute commands, line 4-5). In the same vein, several of calculated results could be stored back to memory as well (N commands, line 7). Further, based on placement and cardinality of PIM compute units, several such computations (across different channels/banks/sub-arrays) can be orchestrated in parallel.

4.3 Ordering Requirement for PIM Commands

With the fine-grained PIM offloading that we envision, the host accelerator (GPU in this work) issues PIM memory instructions (akin to loads, stores) to accomplish the PIM computation. We refer to this host executed kernel (GPU parlance) as *PIM kernel* (Figure 4). GPUs rely on fine-grain hardware scheduling of many threads which is particularly useful in issuing PIM memory operations to several generic PIM compute units that can be placed within the memory hierarchy. PIM memory instructions issued by the host get translated into fine-grained PIM commands discussed in Section 4.2 at the memory controller.

As with existing memory instructions, PIM instructions issued by the host also have ordering requirements. For example, in Figure 4, loading/fetching *vectors a* and *b* should happen before the computation operation(s), which in turn should happen before the store operations for *vector c*. In conventional host code, this ordering is maintained by honoring register dependences. But when the host offloads this computation to a simple PIM unit executing fine-grain commands, it loses its ability to enforce these dependences. PIM instructions issued by the host can be re-ordered to target several performance optimizations. For example, instructions can get re-ordered within the host pipeline, in the network from the host to the memory controller, within the memory controller transaction queue and more. As such, any required order between PIM instructions has to be explicitly enforced.

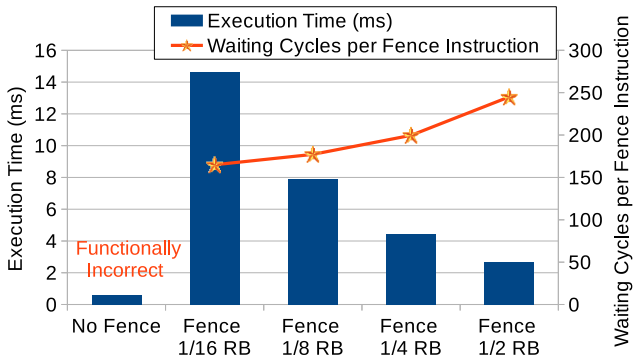

Figure 5: Fence overhead for vector_add kernel.

Host computations rely on memory ordering primitives, aka fences, to ensure ordering of memory instructions. Fences are inserted in the host PIM kernel in Figure 4 to ensure ordering for PIM execution. However, fences are impractical from a performance point of view and insufficient functionally for fine-grained PIM operations[4]. First, fences have considerable performance overheads and are generally used sparingly and judiciously. Fine-grained PIM offloading, however, considerably increases the number of needed fence instructions leading to severe overheads. With more storage, a larger number of PIM instructions (N) can be issued (lines 2, 4, and 7) before issuing a fence. Note that the size of the temporary storage (N) determines the number of loop iterations required to complete the task in Figure 4. Each iteration requires 3 fences; if N is high, fewer iterations and fences are required. For a range of values of N (as a fraction of row-buffer size), fences can slow down execution by 4.5× to 25× as depicted in Figure 5 (see Section 6 for methodology details). Second, for memory instructions which do not return data to the host (stores), existing fences only ensure ordering up until the global serialization point (coherence directory or memory controller) which is insufficient for PIM instructions as they have to be issued to the memory module by the memory controller in the desired order for correctness. As such, existing memory ordering primitives remain a strong impediment to support of FGO/FGA PIM designs. Addressing this challenge is the key focus of this work.

5 ORDERLIGHT DESIGN

In this section we discuss our proposed approach to tackle the challenge of providing efficient memory ordering while enabling FGO/FGA PIM designs.

5.1 Key Insight: Memory-centric Ordering Enforcement

In order to design an efficient memory ordering primitive for fine-grained PIM approaches, we make the key observation that the

[4]Some CPU implementations offer strongly-ordered uncacheable memory instructions. However, they suffer from similar deficiencies as fences[19].

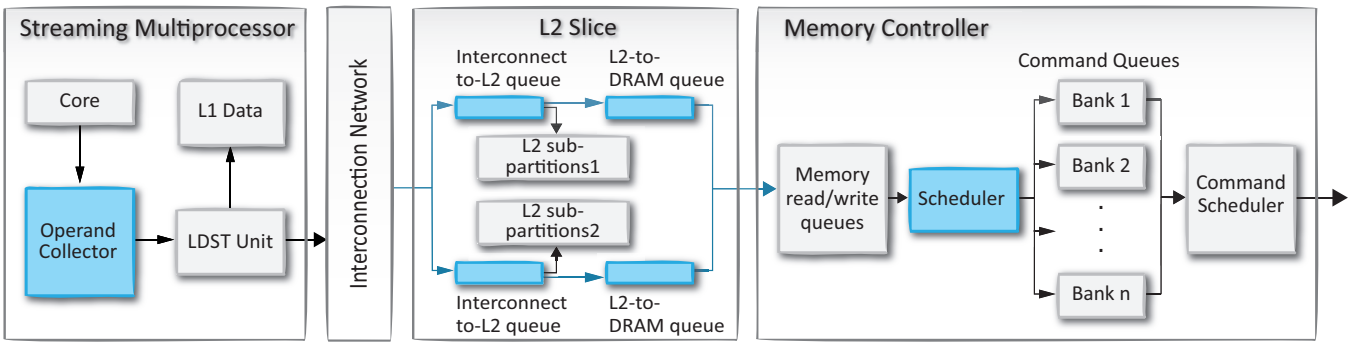

Figure 6: Core and memory pipe in GPU. Modules which re-order are highlighted in blue.

core-centric nature of existing memory ordering primitives is **neither necessary nor sufficient** for PIM instructions. Consider a load instruction followed by a fence instruction. Fence semantics have to ensure that any subsequent memory instructions happen after the load is complete. Completion point for load is at the core (core receives requested data block) and the core incurs wait cycles to ensure this ordering. In contrast, PIM instructions (e.g., *Fetch-and-Add*) are completed at the PIM compute unit. This provides a unique opportunity to push ordering enforcement for PIM instructions to the memory controller, thus, freeing the core to issue PIM instructions without incurring wait cycles. Thus, PIM instructions need *memory-centric* ordering enforcement.

Furthermore, existing memory ordering primitives enforce ordering until the global serialization point (coherence directory or memory controller). However, as PIM instructions are completed at the PIM compute unit, ensuring proper ordering necessitates that the corresponding PIM commands are issued to memory subject to the ordering constraints (to avoid re-ordering at the memory controller). This further makes a case for memory-centric ordering enforcement.

In essence, the host is aware of register dependences and it is trying to communicate these dependences to the memory controller with a lightweight mechanism. Such a lightweight mechanism can out-perform a baseline where the core is responsible for enforcing fences and inevitably suffers from frequent core ⟷ memory latencies.

5.2 OrderLight Overview

To overcome the shortcomings of core-centric memory ordering primitives and exploit the opportunities of memory-centric ordering, we propose the *OrderLight* memory ordering primitive. In our proposed design, in order to express ordering between PIM instructions, the programmer employs the novel *OrderLight* instruction instead of a regular fence. On encountering this instruction, the core generates an *OrderLight* packet, which percolates all the way to the memory controller. The relative ordering of *OrderLight* packet and PIM instructions is maintained at every step of the memory pipe until the PIM requests reach the memory controller. This packet is also preserved by the memory controller in its transaction queue to enforce ordering amongst PIM commands. Finally, as ordering is

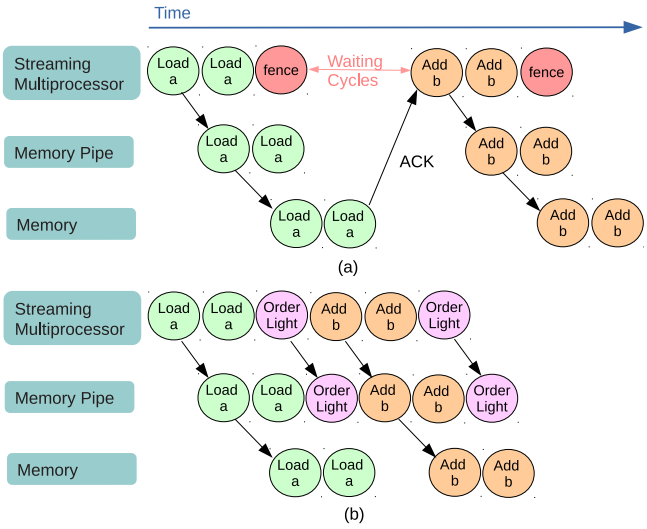

Figure 7: Behavior of fence versus OrderLight: (a) fence keeps the host (SM in GPU) stalled to enforce ordering, (b) OrderLight packet ensures ordering at the memory controller by percolating through the memory pipe obviating stalls at host.

conveyed to and enforced at the memory controller, the core does not incur any wait cycles and can issue PIM instructions unabated.

We highlight the benefits of an *OrderLight* primitive in comparison to a fence in Figure 7. We show the ordering of *Load* and *Fetch-and-Add* instructions of the *vector_add* PIM kernel discussed in Section 4.2. With existing fence primitives, we observe that the core incurs wait cycles (165 to 245 cycles as depicted in Figure 5) to ensure the ordering requirement. In contrast, using *OrderLight* instructions, the *OrderLight* packets are delivered in-order to the memory controller which enforces memory ordering. As such, the host does not incur wait cycles and can issue PIM instructions with high throughput.

5.3 Architectural Changes for OrderLight

In the following sections, we describe the architectural changes needed to realize memory-centric ordering using our proposed

OrderLight primitive in the context of GPUs as host accelerators. Figure 6 depicts a typical GPU architecture along with core and memory pipe (path from core/SM to memory controller) that we consider in this work. We first discuss the design changes needed for our new *OrderLight* instruction in the core and then highlight the changes needed to support the *OrderLight* packet in the memory pipe.

5.3.1 Design Changes in the Core

OrderLight Instruction and OrderLight Packet: Our proposed work introduces a new *OrderLight* instruction which is employed by the programmer to express memory ordering between PIM instructions. An *OrderLight* instruction inserts an *OrderLight* packet into the memory pipe. Figure 8 shows the different fields in an *OrderLight* packet, with example bit widths. The packet is distinguished from normal load/store requests using a 2-bit packet ID. A channel ID (shown as a 4b field in Figure 8) identifies the memory channel for which ordering is to be enforced. The next field is an optional 4b memory-group ID. A memory-group can be a subset of banks, an HBM stack[5], a subset of sub-arrays, etc. Memory-group ID helps enforce ordering for a particular memory-group only. For example, if PIM data structures are mapped to one memory-group and non-PIM data structures are mapped to a different memory-group, the memory-group ID in the *OrderLight* packet informs the architecture to not constrain non-PIM requests whenever possible as they need not be ordered. The *OrderLight* packet can be extended to support ordering across multiple memory-groups (e.g., when operating on partial results from two different PIM kernels) via additional 4b memory-group ID fields. Finally, the fourth field is the packet number within a channel and memory-group. This helps the memory-controller to perform sanity checks and collect statistics.

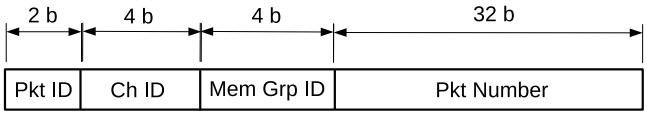

Figure 8: Different fields in an OrderLight packet.

Operand Collector: The operand collector logic in the core schedules operand access to a multi-banked register file. It consists of multiple collector units, each of which collects operands from the register file for one instruction. After all the operands are buffered, an instruction is ready to be issued. Each memory instruction is allocated a collector unit and its register access requests are queued in an arbitration logic block. The arbitration logic can issue register accesses out of order to schedule accesses to multiple banks in parallel.

Fences halt execution before instructions are sent to the operand collector. However, to avoid instruction reordering with *OrderLight* primitive, the *OrderLight* packet is issued only after all preceding PIM requests have left the operand collector. This is achieved by keeping a count of the number of PIM requests residing in the operand collector. The count is incremented every time a PIM request is allocated a collector unit and is decremented when the request is issued. An *OrderLight* packet does not need to go through the operand collector phase and is issued to the LDST queue when the count reads zero. Thus, instruction issue is halted by the *OrderLight* packet for a much shorter period of time in comparison to normal fences. A separate counter is used for each memory-channel and memory-group. To reduce the number of counters, an implementation may limit the number of channels/memory-groups that can be controlled per SM.

[5]HBM groups vertically-stacked memory dies into groups of 4 referred to as *stacks*, somewhat analogous to ranks in DDR memory systems

5.3.2 Design Changes in the Memory Pipe

Caches: Caches contribute to reordering due to cache hits for later requests. A memory request that experiences a cache hit is serviced faster. However, PIM computation requests are meant to reach the memory and do not affect the cache. We consider PIM requests to behave the same way non-temporal loads and stores do. Thus, these requests bypass the caches and are directed to the main memory. For L1 cache, the requests move from the LDST queue to the interconnect network. For L2 cache, the requests move from the interconnect-to-L2 queue to the L2-to-DRAM queue.

Diverging Paths in the Memory Pipe: The memory pipe may consist of one or more diverging paths. For example, many architectures have multiple sub-partitions per L2 slice and separate input/output queues for each sub-partition. L2 sub-partitions are often used to cater to different memory-groups within a memory channel. PIM requests navigated to different sub-partitions may merge later in the memory pipe out of order.

To maintain ordering through divergent paths, we use a *copy-and-merge* technique for the *OrderLight* packet, as shown in Figure 9. When an *OrderLight* packet reaches a divergence point in the memory pipe, the packet is copied into multiple packets that traverse each of the relevant memory sub-paths. For example, if a PIM kernel is issuing PIM requests to a particular memory-group within a channel, and the requests of the memory-group traverse through two of four L2 sub-partitions, then the *OrderLight* packet is copied to generate two *OrderLight* packets, each of which traverses the queues of the relevant sub-partition. The copied packets are merged at the convergence point in the memory pipe. Any requests that follow the *OrderLight* packets in the different memory sub-paths aren't allowed to proceed until all the *OrderLight* copies are merged and the merged packet moves forward.

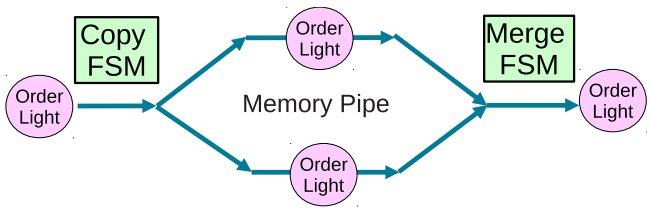

Figure 9: The copy-and-merge technique used for OrderLight packet when divergence is encountered in the memory pipe.

The copy-and-merge technique is achieved using finite state machines (FSM) at divergence and convergence points. The FSM at the divergence point uses information in the *OrderLight* packet,

such as channel ID and memory-group ID, to replicate the packet on each relevant sub-path. The FSM at the convergence point issues a merged packet down the pipe once all copies of the packet are received (number of copies to merge is determined similarly to the divergence point). Note that path divergence among L2 slices does not require re-convergence as each L2 slice is associated with one memory channel with no subsequent merging of paths.

Memory Controller: The memory controller may implement a unified queue or separate queues for read and write requests. It also contains a scheduler which tries to balance increased DRAM efficiency (via techniques such as prioritizing row hits and reducing read/write turnarounds) and fairness. Commands for scheduled requests are enqueued in command queues (which may be separate for each bank), which are then issued to the memory module subject to timing constraints. For separate read and write queues, the copy-and-merge technique has to be adopted, where two *OrderLight* packets are generated and pushed to each of the queues and get merged at the scheduler stage of the controller.

The memory controller is allowed to reorder requests in the following cases: (i) requests aren't separated by an *OrderLight* packet, or (ii) requests belong to different memory-groups. The scheduler is augmented with a request counter and an *OrderLight* flag for each PIM memory-group. The counter associated with a memory-group is incremented when a request to that memory-group is dequeued by the scheduler and decremented when it is scheduled. When the scheduler receives an *OrderLight* packet, the *OrderLight* flag for the appropriate memory-group is set. Any subsequent request to that memory-group is not scheduled until the flag is unset. The flag is unset when the counter for the memory-group is decremented to zero (i.e., all requests preceding the *OrderLight* packet have been scheduled). Once the *OrderLight* flag is reset, the scheduler is free to process subsequent requests.

5.4 Programmability

As discussed in Section 4.3, unlike coarse-grain PIM approaches, programming for fine-grained PIM approaches necessitates the host to execute a PIM kernel comprising of a stream of PIM instructions. In the long-term, we envision compile-time software tools that can automate the generation of PIM instructions based on specially-annotated regions of code expressing computation to be executed on PIM units. In the near-term, intrinsics-like low level primitives can be embedded in high-level code that generate the appropriate fine-grain PIM instructions when compiled. The fields for an *OrderLight* packet such as channel ID and memory-group ID can be populated at compile time by exposing the memory organization. Note that this can also be done in hardware by keeping track of physical addresses to which prior PIM requests are issued.

In a GPU-like host processor with many cores, hardware contexts, and concurrent threads, a subset of the cores or hardware contexts can be set aside for executing the PIM kernel. Further, to reduce the need for synchronization among host software warps, the control of each PIM unit (or a set of PIM units) can be limited to a single host warp. In other words, each PIM unit receives PIM instructions from a single host warp, avoiding the need for synchronization among multiple host warps in orchestrating PIM computation. We utilize such a model in our evaluations.

As with any PIM computation, FGO/FGA requires keeping data in different levels of the memory hierarchy coherent. For example, dirty data (of PIM operands) should be flushed to the main memory before PIM computation on that data is invoked, and data updated by PIM computation should be invalidated in the host's caches. Both of these aspects are orthogonal to the granularity of PIM offloading and arbitration, and is similar to previous PIM research. The application could issue (selective) cache flushes before launching a PIM kernel to ensure a consistent view of memory or the PIM architecture may support such functionalities as part of the PIM instructions.

This work assumes a hierarchical/scoped, relaxed consistency model like current GPUs. PIM operations are similar to system-scope but have even stronger requirements as the operations need to go all the way out to DRAM, not just the level of memory that is visible to all units of the system (i.e., the memory controller or a global coherence point). Thus, *OrderLight* fits within the scoped/hierarchical consistency model with no changes that affect the non-PIM operations of the system.

6 METHODOLOGY

We use GPGPU-Sim [5] to evaluate the performance impact of our proposed *OrderLight* primitive when used with a GPU host. The GPU micro-architecture and the memory parameters assumed in this work are summarized in Table 1.

GPU Parameters			
GPU Model:		Volta Titan V	
Number of SMs:	80	Core Frequency:	1200 MHz
L1 Data Size:	32 KB	Shared Memory Size:	96 KB
L2 Size:	3 MB	L2 Queue Size:	64
Memory Scheduler:	FRFCFS	R/W Queue Size:	64
Interconnect to L2 latency:	120 cycles	L2 to DRAM scheduler latency:	100 cycles
Memory Parameters			
Memory Model:		HBM	
Memory Channels:	16	DRAM Bus Width:	32B
Banks per Channel:	16	Memory Frequency:	850 MHz
Memory Timing: (in cycles)	CCD=1:RRD=3:RCDW=9:RAS=28:RP=12: CL=12:WL=2:CDLR=3:WR=10:CCDL=2:WTP=9		

Table 1: Simulator details.

Workloads: PIM is useful for data-intensive applications which have low compute-to-memory ratio. As such, we first analyze the behavior of *OrderLight* and fence using the stream benchmark [37] which is representative of many kernels in GPGPU applications such as feature_map addition, scalar_product, activation_functions, etc. Next, we evaluate the performance improvement due to *OrderLight* on a set of GPGPU kernels in machine learning, data analytics, and genomic applications, discussed in Section 2. Table 2 shows a summary of our workload suite which represents a range of different compute-to-memory ratios.

Modelling PIM kernels: For our evaluation, we write PIM kernel (CUDA code) for each workload so as to mimic the sequence of PIM instructions, similar to the pseudo-code shown in Figure 4. As with any PIM architecture, writing the code requires knowledge of the memory organization, such as the interleaving granularity of

Kernels	Description	Compute: Memory Ratio	More than one data structure accessed?
Stream Benchmark			
Scale	a[i] = scalar*a[i]	1:1	No
Copy	b[i] = a[i]	0:2	Yes
Daxpy	b[i] = b[i] + scalar*a[i]	2:2	Yes
Triad	c[i] = a[i] + scalar*b[i]	2:3	Yes
Add	c[i] = a[i] + b[i]	1:3	Yes
Other Workloads			
BN_Fwd [22]	Batch Normalization Forward Phase	7:3	Yes
BN_Bwd [22]	Batch Normalization Backward Phase	14:6	Yes
FC [31]	Fully Connected	2:1	No
KMeans [8]	KMeans Clustering	10:1	No
SVM [40]	Support Vector Machine	2.5:2	Yes
Hist [41]	Histogram	3:2	Yes
Gen_Fil [28]	Genomic Sequence Filtering (GRIM Algo)	3:1	No

Table 2: Summary of workloads.

physical memory across channels, size of PIM temporary storage (Section 4.1), etc.

The PIM kernels use one warp per memory channel and utilize the SIMT parallelism to generate N PIM instructions in parallel (lines 2, 5, and 8 in Figure 4). Our evaluation shows that using one SM per two warps is sufficient to execute the PIM kernel (8 SMs for 16 memory channels), thereby apportioning sufficient SMs to run compute-bound kernels in parallel. We use non-temporal store instructions to mimic PIM instructions. Similar to most fine-grained PIM instructions, non-temporal stores bypass the host caches and avoid allocating the lines in the caches. We assume that the GPU driver allocates large pages for the PIM data structures and ensure that all of the operands needed for a PIM computation align within the memory regions associated with each PIM unit.

Baseline Limitations: The baseline fence functionality is implemented to ensure that memory requests are issued to the memory before proceeding with subsequent instructions. There is limited scope in hiding the fence latency with higher parallelism. This is because multiple warps sending instructions to the same PIM unit would require warp-level synchronization to have a deterministic instruction order (otherwise we are at the mercy of the warp scheduler). This software synchronization would enforce only one warp talking to one PIM unit at any point in time.

The downside of using a fence is that independent non-PIM instructions following the fence will be blocked. Although PIM kernels are mostly a sequence of fine-grained PIM instructions, there may be a few independent instructions (e.g., address calculation for operands). Note that such instructions are rare (PIM kernels may use PIM instructions with offset from a base address) and executed in the GPU core within a few cycles. Thus, executing such instructions when the core is idling on a fence does not help.

Since fence instructions keep the core idle, multiple PIM warps can be executed per SM via context switching. Our observations show up to eight warps can be executed per SM, thereby requiring two SMs to send commands to 16 channels. Note that more compute is needed (up to eight SMs) to support the high command throughput facilitated by OrderLight.

Evaluation Metrics: In the results section, we use two new metrics: (i) PIM Command Bandwidth (GigaCommands/s or GC/s), which represents the number of PIM commands sent to the memory per second, and (ii) PIM Data Bandwidth (GB/s), which represents the bandwidth at which PIM processes data within the memory module. Note that the PIM Data Bandwidth reflects the product of PIM command bandwidth and the bandwidth multiplication factor (BMF) over host bandwidth. PIM Command Bandwidth highlights the gains using *OrderLight* in the throughput of sending PIM commands, which in turn provides gains in PIM Data Bandwidth.

7 RESULTS

In this section, we first discuss benefits of using *OrderLight* for stream kernels, followed by its benefits for a suite of key computations from modern applications.

7.1 OrderLight Speedups for Stream Benchmark

Figure 10a shows the improvement in PIM command bandwidth and data bandwidth for four different temporary storage (TS) sizes for each of the stream kernels when using fence versus the *OrderLight* primitive (with a BMF of 16). Figure 10b shows a similar comparison but for execution time and the number of stall cycles at the core.

7.1.1 Improved Command Bandwidth

Ordering primitives slow down the rate at which PIM commands are sent to the memory modules due to ordering constraints which in turn limits PIM benefits. As such, we study the improvement in command bandwidth that *OrderLight* achieves over existing fence primitives. To do so, we use the Add kernel as an exemplar of other kernels we study.

The Add kernel is representative of the *vector_add* kernel shown in Figure 4. We observe that the command bandwidth when using the *OrderLight* primitive is on average 2.6× higher than when using the fence primitive. This is largely due to elimination of the stall cycles induced by the fence at the core because of which the memory controller is also starved from issuing PIM commands.

Effect of Temporary Storage (TS) near PIM Unit: The size of TS associated with a PIM compute unit dictates two important factors which influence the command bandwidth and we vary this parameter to study its effect.

First, it dictates the number of ordering primitives that are necessary. The larger the TS size, the greater the number of PIM instructions that can be sent before issuing an ordering primitive; when copying data from memory to TS prior to performing computation on that data, no ordering is necessary among multiple independent copy commands. Ordering is only required at the end of the copy commands before computation is performed on the data now in TS. The decreasing number of fences with increasing TS size improves the command bandwidth for fences as evident in Figure 10a. This factor has negligible impact on *OrderLight* as it does not pay a significant penalty of wait cycles at the core per *OrderLight* instruction.

Second, TS also dictates the peak achievable command bandwidth. This is so, as it dictates the DRAM locality. Consider again the Add kernel. It accesses three different vectors, a, b, and c, each of which get mapped to a different DRAM row due to the memory

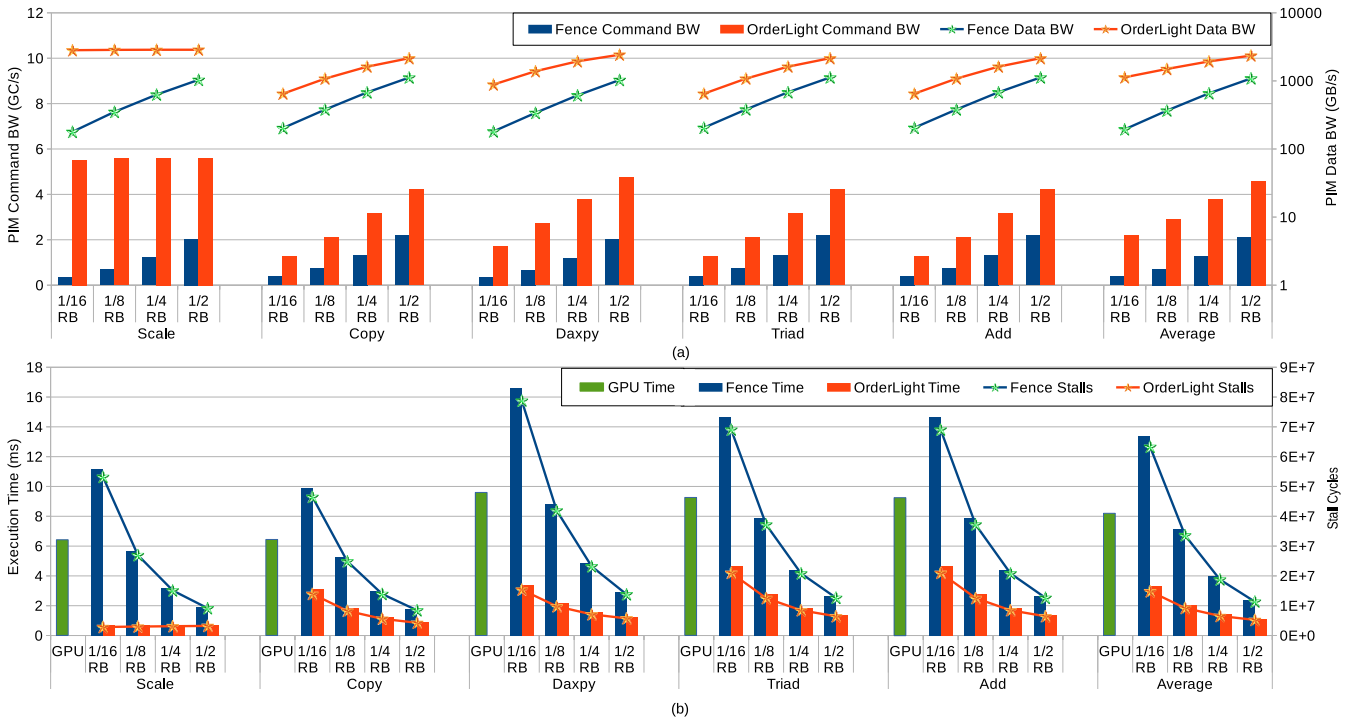

Figure 10: Comparison of fence versus *OrderLight* for stream benchmarks using the following metrics: (a) PIM command bandwidth (linear scale) and data bandwidth (log scale), and (b) execution time and the number of stall cycles at the core.

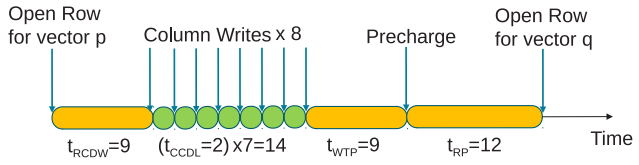

Figure 11: DRAM timing for opening the DRAM row for vector p, sending 8 write requests (equivalent to PIM commands), followed by opening the row for vector q.

mapping policy. The kernel issues a series of instructions to one vector before switching to the next vector (Figure 4). Figure 11 shows the timing constraints for a similar example that writes two vectors (p and q). For a TS sized to hold 256B of data from p, the memory controller can issue 8 row-hit accesses (32B each) for vector p before switching to a different row for vector q, within a period of 44 cycles. The latency of opening and closing a row ($t_{RCDW} + t_{WTP} + t_{RP}$) limits the command bandwidth. The achievable peak command bandwidth in this case is 2.3 GC/s (8/44 * peak). We observe in Figure 10a that, while the command bandwidth achieved with fences is far below that of peak, *OrderLight* comes close to this peak (2.1 GC/s).

Effect of the Number of Memory Operands: An equally important factor that also affects peak command bandwidth is the number of operands read or written from/to memory by a computation as different operands typically map to different DRAM rows. As an example, Scale benchmark only works on a single DRAM row in comparison to Add benchmark which works on three DRAM rows. Consequently, the overheads of row open/close is much lower for the Scale benchmark, increasing its achievable peak command bandwidth. Further, as Scale only works on a single DRAM row, TS (and its size) is immaterial for this benchmark.

Overall, across kernels, *OrderLight* considerably improves the command bandwidth realized for PIM commands and often reaches peak command bandwidth possible.

7.1.2 *Improved Data bandwidth*

Figure 10a also shows the data bandwidth offered by PIM. Recall that, memory bandwidth available for PIM compute units is often much higher than that available to the host and is the key benefit of PIM. We see that the data bandwidth with *OrderLight* is higher than the peak external data bandwidth of the memory module (405 GB/s) by 4.3× on average and outperforms the data bandwidth with fence by 3.8×.

7.1.3 *Improved PIM Speedup over Host*

Figure 10b shows that the use of fences slows down PIM execution drastically to show little to no benefits over GPU execution (green bars in the graph). PIM execution with fences shows improvement over GPU only when using larger TS (1/4 or 1/2 of row-buffer) by 2× to 3.4×. On the other hand, *OrderLight* consistently outperforms GPU execution time for every TS size by 3.5× to 7.4× on average.

The key reason for improved PIM speedups is attributed to reduced stall cycles at the core in sending PIM instructions which in turn leads to improved command bandwidth. This is depicted in

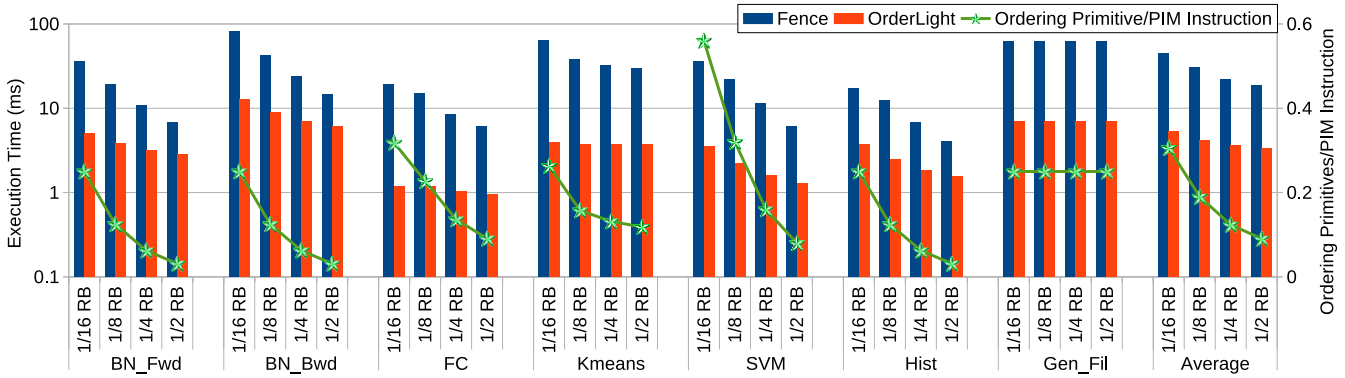

Figure 12: Improvement in execution time with the OrderLight primitive over fence for a set of data-intensive computations in GPGPU applications. The graph also plots the number of ordering primitives issued per PIM instruction for each kernel.

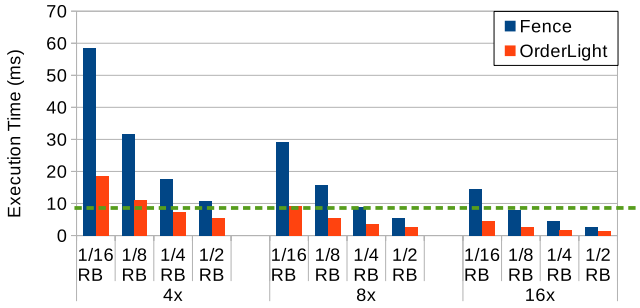

Figure 13: Comparison of fence versus OrderLight using different Bandwidth Multiplication Factor for the Add kernel.

Figure 10b which shows that the number of stall cycles for the two different primitives closely resemble the plot for execution time. When using *OrderLight*, the number of stall cycles decreases with bigger *TS* because the memory controller can issue commands at a faster rate, thereby decreasing backward pressure on queues in the memory pipe.

7.1.4 *Speedups for Varied PIM Solutions*

As discussed in Section 4.1, by varying placement and cardinality of PIM compute units, we can study benefits of *OrderLight* for varied PIM solutions. We study three different BMF (bandwidth multiplication factor) for PIM over host in Figure 13 using the Add kernel. We observe in this figure that *OrderLight* consistently performs better than fence by 1.9× to 3.1×. In fact, the performance with fence is worse or comparable to GPU execution in 8 of the 12 cases, whereas *OrderLight* provides improvement over GPU in 10 of the 12 cases. This is because, with decreasing BMF, a greater number of PIM commands has to be sent to the memory to perform the same job, which increases the burden with fence.

7.2 Speedups for Data-Intensive Applications

Figure 12 shows the performance improvement with *OrderLight* primitive over fence for a set of important data-intensive computations from GPGPU applications. We observe that *OrderLight* provides 5.5× to 8.5× improvement in execution time over the fence primitive for the suite of computations evaluated, and considerably improves PIM speedups over host as compared to the fence primitive.

From Table 2 we see that FC and Kmeans access only one data structure per computation, which is why they experience more row locality than the other kernels. For the same reason, they show little variability in performance with different *TS* size when using the *OrderLight* primitive. Gen_Fil issues irregular PIM requests on 128-B data granularity (1/16th of RB). This is why Gen_Fil shows no variability with bigger *TS* size for both fence and *OrderLight* primitives.

We observe that FC, Kmeans, and Gen_Fill kernels show high improvement in execution time with *OrderLight* even for bigger *TS* size. As shown in Figure 12, the number of ordering primitives issued per PIM instruction decreases with increase in *TS* size at a much slower rate for these kernels in comparison to the other kernels (rate of decrease: FC: 33%, Kmeans: 22%, Gen_Fil: 0%, Others: 50%). This is because of the computation structure of these kernels. Thus, a lot more ordering primitives are issued even for bigger *TS* sizes, which hurts the performance when using fences.

8 RELATED WORK

In this section, we briefly discuss some FGO/FGA PIM designs from the literature and also contrast *OrderLight* with fences for persistent memory ordering which, at a high-level, appear related.

8.1 Fine-grain Offload and Fine-grain Arbitration PIM

Although alternate (i.e., non-FGO/FGA) PIM designs have merits as discussed in Section 3, FGO/FGA PIM designs are particularly suited for modern workloads with compute and data-intensive phases. These designs are also compatible with mainstream memory interfaces. These and other characteristics discussed in Section 3.5 make them particularly desirable in the current computing landscape. Consequently, we focus on research works which fall under this sub-class here.

FIMDRAM [32] and Lee et al. [34] focus on designing a PIM architecture for machine learning that can be finely interleaved with commodity DRAM command behaviors. Other works have

used PIM operations using slight variants of existing DRAM operations [14, 45] or instruction-granularity offload of application-specific operations to PIM [3, 39]. However, these efforts focus on the PIM architectures and do not address the requirements for efficient host-side control of PIM, which is our focus in this work. As a result, *OrderLight* is applicable for these (and more) FGO/FGA PIM designs and stands to complement them by providing efficient PIM operation ordering capabilities.

Kim et al. [27] use sequence numbers to order the sequence in which PIM operands are processed. However, this requires complex deadlock prevention logic such as credit-based buffer management. Additionally, the latency of sending credit reservation requests from SMs and receiving acknowledgments may reduce the PIM command bandwidth. This approach also requires a fair bit of buffering at the memory which can be expensive when adding PIM to a commodity DRAM. *OrderLight* on the other hand uses lightweight packets which make the implementation simpler, does not cause deadlocks, and can be implemented on existing DRAM interfaces that do not have credit/op ID semantics.

8.2 Memory Ordering for Persistence

A key differentiating factor for memory ordering guarantees needed for FGO/FGA PIM designs compared to existing fences is that the order has to be enforced at the PIM units, and not at the processor cores. At a high-level, systems with persistent, non-volatile memory (NVM) also require that writes "at memory" appear to have occurred in the specified order to ensure recoverability of persistent data structures. Research works such as Delegated Persist Ordering (DPO) [30] track core-centric constructs such as intra-thread and inter-thread dependencies, coherence traffic, etc. using persist buffers and bloom filters in traditional CPUs to offload epochs of persistent writes to the NVM controller. However, DPO and several such proposals in this domain [10, 36] rely on enforcing this order in a core-centric fashion which is both overly constrained and unnecessary for PIM.

9 CONCLUSION

With crucial applications exhibiting both compute and data-intensive phases, it is imperative that accelerators be coupled with PIM-enabled solutions. To that end, in this work we first introduced a taxonomy to better understand the design space of an accelerator (e.g., a GPU) interacting with PIM-enabled memory when considering the temporal granularity of both computation offloads to near-memory logic and arbitration of PIM and host memory accesses. Based on this taxonomy, we observe that prior PIM proposals largely focus on coarse-grain approaches which can have steep costs. On the other hand, while fine-grain PIM approaches avoid these costs, a key impediment to realizing them is support for efficient memory ordering primitives for fine-grained PIM instructions. As such, we propose a novel memory-centric ordering primitive, *OrderLight*, which overcomes the shortcomings of traditional core-centric memory ordering primitives such as fences. Evaluations based on key computations from several application domains show that *OrderLight* delivers 5.5× to 8.5× speedup over traditional fences.

The innovations discussed in the paper are broadly applicable to other hosts, including out-of-order (OoO) CPUs. Even though OoO cores can hide memory access latencies via ILP/MLP, fence overheads can still be in the order of 100 cycles [6]. OoO cores have renaming units and reservation stations, where ordering should be maintained similar to Operand Collector in GPUs. Network-on-Chip (NoCs) between different levels of the cache hierarchy may unorder PIM requests – ideas related to path divergence are applicable here.

FGO/FGA based PIM architectures are relatively new. Recent papers [32–34] explore the device aspect of such architectures. This paper explores the systems aspect of such architectures from the host's perspective. We believe future work will explore efficiently utilizing such PIM primitives (programmability, co-scheduling, compilers, etc).

ACKNOWLEDGMENTS

We thank the anonymous reviewers and the shepherd, whose comments/suggestions helped improve this paper.

REFERENCES

[1] Shaizeen Aga, Nuwan Jayasena, and Mike Ignatowski. 2019. Co-ML: A Case for Collaborative ML Acceleration Using near-Data Processing. In *Proceedings of the International Symposium on Memory Systems*. Association for Computing Machinery, 506–517.
[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *Proceedings of ISCA-42*. 105–117.
[3] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *Proceedings of ISCA-42*. 336–348.
[4] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. 2016. Comparative Study of Caffe, Neon, Theano, and Torch for Deep Learning. *arXiv preprint arXiv:1511.06435* (2016).
[5] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of ISPASS*. 163–174.
[6] Milewski Bartosz. 2008. Who ordered memory fences on an x86? "https://bartoszmilewski.com/2008/11/05/who-ordered-memory-fences-on-an-x86/".
[7] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T Malladi, Hongzhong Zheng, and Onur Mutlu. 2016. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-In-Memory. *IEEE Computer Architecture Letters* 16, 1 (2016), 46–50.
[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of IISWC*. 44–54.
[9] Benjamin Y Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez. 2020. Near Data Acceleration with Concurrent Host Access. In *Proceedings of ISCA-47*. 818–831.
[10] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-Addressable, Persistent Memory. In *Proceedings of SOSP*. 133–146.
[11] Fabrice Devaux. 2019. The True Processing-In-Memory Accelerator. Hot Chips-31. , 24 pages.
[12] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, et al. 2002. The Architecture of the DIVA Processing-In-Memory Chip. In *Proceedings of ICS*. 14–25.
[13] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules. In *Proceedings of HPCA-21*. 283–295.
[14] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. 2019. ComputeDRAM: In-Memory Compute Using Off-The-Shelf DRAMs. In *Proceedings of MICRO-52*. 100–113.
[15] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *Proceedings of ASLPOS-22*. 751–764.
[16] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in Memory: The Terasys Massively Parallel PIM Array. *IEEE Computer* 28, 4 (1995), 23–31.

[17] Qing Guo, Xiaochen Guo, Ravi Patel, Engin Ipek, and Eby G Friedman. 2013. AC-DIMM: Associative Computing with STT-MRAM. In *Proceedings of ISCA-40*. 189–200.
[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. In *Proceedings of the European Conference on Computer Vision*. Springer, 630–645.
[19] Richard Gerard Hofmann, Thomas Andrew Sartorius, Thomas Philip Speier, Jaya Prakash Subramaniam Ganasan, James Norris Dieffenderfer, and James Edward Sullivan. 2015. Enforcing Strongly-Ordered Requests in a Weakly-Ordered Processing. https://patents.google.com/patent/US9026744B2/
[20] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *Proceedings of ISCA-43*. 204–216.
[21] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation. In *Proceedings of ICCD-34*. 25–32.
[22] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of ICML-32*. 448–456.
[23] JEDEC. 2019. High Bandwidth Memory DRAM (HBM1, HBM2). "https://www.jedec.org/standards-documents/docs/jesd235a".
[24] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of ISCA-44*. 1–12.
[25] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. 1999. FlexRAM: Toward an Advanced Intelligent Memory System. In *Proceedings of ICCD*. 192–201.
[26] Duckhwan Kim, Jae Ha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *Proceedings of ISCA-43*. 380–392.
[27] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. 2017. Toward Standardized Near-Data Processing with Unrestricted Data Placement for GPUs. In *Proceedings of SC*. 1–12.
[28] Jeremie S Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. 2018. GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-In-Memory Technologies. *BMC Genomics* 19, 2 (2018), 23–40.
[29] Peter M Kogge. 1994. The EXECUBE Approach to Massively Parallel Processing. In *Proceedings of ICPP-23*. 77–84.
[30] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M Chen, and Thomas F Wenisch. 2016. Delegated Persist Ordering. In *Proceedings of MICRO-49*. 1–13.
[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of NIPS-26*. 84–90.
[32] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, et al. 2021. 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 *Proceedings of ISSCC*, Vol. 64. 350–352.
[33] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, Seongil O, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology. In *Proceedings of ISCA-48*. 43–56.
[34] Won Jun Lee, Chang Hyun Kim, Yoonah Paik, Jongsun Park, Il Park, and Seon Wook Kim. 2019. Design of Processing-"Inside"-Memory Optimized for DRAM Behaviors. *IEEE Access* 7 (2019), 82633–82648.
[35] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *Proceedings of MICRO-50*. 288–301.
[36] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-Ordering Consistency for Persistent Memory. In *Proceedings of ICCD-32*. 216–223.
[37] John D McCalpin et al. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society TCCA Newsletter* 2 (1995), 19–25.
[38] Micron. 2018. Hybrid Memory Cube Specification 2.0. https://www.micron.com/-/media/client/global/documents/products/data-sheet/hmc/gen2/hmc_gen2.pdf
[39] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *Proceedings of HPCA-23*. 457–468.
[40] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. 2006. Minebench: A Benchmark Suite for Data Mining Workloads. In *IEEE International Symposium on Workload Characterization*. 182–188.
[41] NVIDIA. 2020. NVIDIA CUDA SDK 4.2. "https://developer.nvidia.com/cuda-toolkit-archive".
[42] Mark Oskin, Frederic T Chong, and Timothy Sherwood. 1998. Active Pages: A Model of Computation for Intelligent Memory. In *Proceedings of ISCA-25*. 192–203.
[43] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads. In *Proceedings of ISPASS*. 190–200.
[44] Andreas Schlapka. 2018. Micron Announces Shift in High Performance Memory Roadmap Strategy. https://www.micron.com/about/blog/2018/august/micron-announces-shift-in-high-performance-memory-roadmap-strategy.
[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. 2017. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *Proceedings of MICRO-50*. 273–287.
[46] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *Proceedings of ISCA-43*. 14–26.
[47] Hyunsung Shin, Dongyoung Kim, Eunhyeok Park, Sungho Park, Yongsik Park, and Sungjoo Yoo. 2018. McDRAM: Low Latency and Energy-Efficient Matrix Computations in DRAM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2613–2622.
[48] Yatish Turakhia, Kevin Jie Zheng, Gill Bejerano, and William J. Dally. 2018. Darwin: A Hardware-Acceleration Framework for Genomic Sequence Alignment. In *Proceedings of ASPLOS-23*. 199–213.
[49] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, et al. 2017. SCALEDEEP: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. In *Proceedings of ISCA-44*. 13–26.