Near Data Acceleration with Concurrent Host Access

Benjamin Y. Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez
The University of Texas at Austin
{bjcho,yongkee.kwon,sklym,mattan.erez}@utexas.edu

***Abstract*—Near-data accelerators (NDAs) that are integrated with the main memory have the potential for significant power and performance benefits. Fully realizing these benefits requires the large available memory capacity to be shared between the host and NDAs in a way that permits both regular memory access by some applications and accelerating others with an NDA, avoids copying data, enables collaborative processing, and simultaneously offers high performance for both host and NDA. We identify and solve new challenges in this context: mitigating row-locality interference from host to NDAs, reducing read/write-turnaround overhead caused by fine-grain interleaving of host and NDA requests, architecting a memory layout that supports the locality required for NDAs and sophisticated address interleaving for host performance, and supporting both packetized and traditional memory interfaces. We demonstrate our approach in a simulated system that consists of a multi-core CPU and NDA-enabled DDR4 memory modules. We show that our mechanisms enable effective and efficient concurrent access using a set of microbenchmarks, then demonstrate the potential of the system for the important stochastic variance-reduced gradient (SVRG) algorithm.**

I. Introduction

Processing data in or near memory using *near data accelerators* (NDAs) is attractive for applications with low temporal locality and low arithmetic intensity. NDAs help by performing computation close to data, saving power and utilizing proximity to overcome the bandwidth bottleneck of a main memory "bus" (e.g., [2], [3], [6]–[8], [12], [23], [25], [26], [28], [39], [43], [44], [51], [63], [77]). Despite decades of research and recent demonstration of true NDA technology [7], [21], [56], [60], [65], many challenges remain for making NDAs practical, especially in the context of *main-memory NDA*.

In this paper we address several of these outstanding issues in the context of an NDA-enabled main memory. Our focus is on memory that can be concurrently accessed both as an NDA and as a memory. Such memory offers the powerful capability for the NDA and host processor to collaboratively process data without costly data copies. Prior research in this context is limited to fine-grained NDA operations of, at most, cache-line granularity. However, we develop techniques for coarse-grain NDA operations that amortize host interactions across processing entire DRAM rows. At the same time, our NDA does not block host memory access, even when the memory devices are controlled directly by the host (e.g., a DDRx-like DIMM), which can reduce access latency and ease adoption.

Figure 1 illustrates an exemplary NDA architecture, which presents the challenges we address, and is similar to other recently-researched main-memory NDAs [7], [8], [23]. We choose a DIMM-based memory system because it offers the high capacity required for a high-end server's main memory. Each DIMM is composed of multiple chips, with one or more DRAM dice stacked on top of a logic die in each chip, using a low-cost commodity 3DS-like approach. Processing elements (PEs) and a memory controller are located on the logic die. Each PE can access memory internally through the NDA memory controller. These local NDA accesses must not conflict with external accesses from the host (e.g., a CPU). A rank that is being accessed by the host cannot at the same time serve NDA requests, though the bandwidth of all other ranks in the channel can be used by the NDAs. There is no communication between PEs other than through the host. While not identical, recent commercial NDA-enabled memories exhibit similar overall characteristics [21], [56].

Surprisingly, no prior work on NDA-enabled main memory examines the architectural challenges of simultaneous and concurrent access to memory devices from both the host and NDAs. In this work, we address two key challenges for enabling performance-efficient NDAs in a memory system that supports concurrent access from both a high-performance host and the NDAs.

The first challenge is that interleaved accesses may hurt memory performance because they can both decrease row-buffer locality and introduce additional read/write turnaround penalties. The second challenge is that each NDA can process kernels that consume entire arrays, though all the data that a single operation processes must be local to a PE (e.g., a memory chip). Therefore, enabling cooperative processing requires that host physical addresses are mapped to memory locations (channel, rank, bank, etc.) in a way that both achieves high host-access performance (through effective and complex interleaving) and maintains NDA locality across all elements of all operands of a kernel. We note that these challenges exist when using either a packetized interface, where the memory-side controller interleaves accesses between NDAs and the host, or a traditional host-side memory controller that sends explicit low-level memory commands.

For the first challenge (managing concurrent access), we identify reduced row-buffer locality because of interleaved host requests as interfering with NDA performance. In contrast, it is the increased read/write turnaround frequency re-

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00072

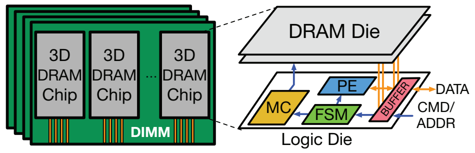


Fig. 1: Exemplary NDA architecture.

sulting from NDA writes that mainly interfere with the host. We provide two solutions in this context. First, we develop a new bank-partitioning scheme that limits interference to just those memory regions that are shared by the host and NDAs, thus enabling colocating host-only tasks with tasks that use the NDAs. This new scheme is the first that is compatible with huge pages and also with the advanced memory interleaving functions used in recent processors. Partitioning mitigates interference from the host to the NDAs and substantially boosts their performance (by $1.5-2\times$).

Second, we control interference on shared ranks by opportunistically issuing NDA memory commands to those ranks that are even briefly not used by the host and curb NDA to host interference with mechanisms that can throttle NDA requests, either selectively when we predict a conflict (*next-rank prediction*) or stochastically.

For the second challenge (NDA operand locality), we enable fine-grain collaboration by architecting a new data layout that preserves locality of operands within the distributed NDAs while simultaneously affording parallel accesses by the high-performance host. This layout requires minor modifications to the memory controller and utilizes coarse-grain allocations and physical-frame coloring in OS memory allocation. This combination allows large arrays to be shuffled across memory devices (and their associated NDAs) in a coordinated manner such that they remain aligned in each NDA. This is crucial for coarse-grain NDA operations that can achieve higher performance and efficiency than cacheline-oriented fine-grain NDAs (e.g., [3], [33], [41]).

An additional and important challenge exists in systems where the host maximizes its memory performance by directly controlling memory devices rather than relying on a packetized interface [29], [65]. Adding NDA capabilities requires providing local memory controllers near memory in addition to the host ones, which introduces a coordination challenge. We coordinate memory controllers and ensure a consistent view of bank and timing state with only minimal signaling that does not impact performance by replicating the controller finite state machines (FSMs) at both the NDA and host sides of the memory channels. Replicating the FSM requires all NDA accesses to be determined only by the NDA operation (known to the host controller) and any host memory operations. Thus, no explicit signaling is required from the NDAs back to the host. We therefore require that for non-packetized NDAs, each NDA operation has a deterministic access pattern for all its operands (which may be arbitrarily fine-grained).

In this paper, we introduce *Chopim*, a SW/HW holistic solution that enables concurrent host and NDA access to main memory by addressing the challenges above with fine temporal access interleaving to physically-shared memory devices. We perform a detailed evaluation both when the host and NDA tasks process different data and when they collaborate on a single application. We demonstrate that Chopim enables high NDA memory throughput (up to 97% of unutilized bandwidth) while maintaining host performance. Performance and scalability are better than with prior approaches of partitioning ranks and only allowing coarse-grain temporal interleaving, or with only fine-grain NDA operations.

We demonstrate the potential of host and NDA collaboration by studying a machine-learning application (logistic regression with stochastic variance-reduced gradient descent [37]). We map this application to the host and NDAs such that the host stochastically updates weights in a tight inner loop that utilizes the speculation and locality mechanisms of the CPU while NDAs concurrently compute a correction term across the entire input data that helps the algorithm converge faster. Collaborative and parallel NDA and host execution can speed up this application by $2\times$ compared to host-only execution and $1.6\times$ compared to non-concurrent host and NDA execution. We then evaluate the impact of colocating such an accelerated application with host-only tasks.

In summary, we make the following main contributions:

- We identify new challenges in concurrent access to memory from the host and NDAs: bank conflicts from host accesses curb NDA performance and read/write-turnaround penalties from NDA writes lower host performance.
- We reduce bank conflicts with a new bank partitioning architecture that, for the first time, is compatible with both huge pages and sophisticated memory interleaving.
- To decrease read/write-turnaround overheads, we throttle NDA writes with two mechanisms: *next-rank prediction* delays NDA writes to the rank actively read by the CPU; and *stochastic issue* throttles NDA writes randomly at a configurable rate.
- We develop, also for the first time, a memory data layout that is compatible with both the host and NDAs, enabling them to collaboratively process the same data in parallel while maintaining high host performance with sophisticated memory address interleaving.
- To show the potential of collaboratively processing the same data, we conduct a case study of an important ML algorithm that leverages the fast CPU for its main training loop and the high-BW NDAs for summarization steps that touch the entire dataset. We develop a variant that executes on the NDAs and CPU in parallel, which increases speedup to 2X.

II. Background

DRAM Basics. A memory system is composed of memory channels that operate independently. In each memory channel, one or more memory modules (DIMMs) share command/address (C/A) and data bus. A DIMM is usually composed of one or two physical ranks where all chips in the same rank operate together. Each chip and thus rank is composed of multiple banks and bank state is independent. Each bank can be in an opened or closed state and, if opened, which row is opened. To access a certain row, the target row must be opened first. If another row is already open, it must be closed before the target row is opened, which is called *bank conflict* and increases access latency. The DRAM protocol specifies the timing parameters and protocol accessing DRAM. These are managed by a per-channel memory controller.

Address Mapping. The memory controller translates OS-managed physical addresses into DRAM addresses, which are composed of indices to channel, rank, bank, row, and column. Typically, memory controllers follow the following policies in their address mapping to minimize access latency: interleaving address across channels with fine granularity is beneficial since they can be accessed independently from each other. On the other hand, ranks are interleaved at coarse granularity since switching to other ranks in the same channel incurs a penalty. In addition, XOR-based hash mapping functions are used when determining channel, rank, and bank addresses to maximally exploit bank-level parallelism. This also minimizes bank conflicts when multiple rows are accessed with the same access pattern since the hash function shuffles the bank address order [84]. To accomplish this, some row address bits are used along with channel, rank, and bank address bits [66].

Write-to-Read Turnaround Time. In general, interleaving read and write DRAM transactions incurs higher latency than issuing the same transaction type back to back. Issuing a read transaction immediately following a write suffers from particularly high penalty. The memory controller issues the write command and loads data to the bus after tCWL cycles. Then, data is transferred for tBL cycles to the DRAM device and written to the cells. The next read command can only be issued after tWTR cycles, which guarantees no conflict on the IO circuits in DRAM. The high penalty stems from the fact that the actual write happens at the end of the transaction whereas a read happens right after it is issued. For this reason, the opposite order, read to write, has lower penalty.

NDA Basics. Near-data accelerators add processing elements near memory to overcome the physical constraints that limit host memory bandwidth. Host peak memory bandwidth is determined by the number of channels and peak bandwidth per channel. Any NDA accesses on the memory side of a channel can potentially increase overall system bandwidth. For example. a memory module with multiple ranks offers more bandwidth in the module than available at the channel. Similarly, multiple banks on a DRAM die can also offer more bandwidth than available off of a DRAM chip. However, because NDAs only offer a BW advantage when they access data in their local memory, data layout is crucial for performance. A naive layout may result in frequent data movement among NDAs and with the host.

Baseline NDA Architecture. Our work targets NDAs that are integrated within high-capacity memory modules such that their role as both main memory and as accelerators is balanced. Specifically, our baseline NDA devices are 3D-integrated within DRAM chips on a module (DIMM), similar to 3DS DDR4 [17] yet a logic die is added. DIMMs offer high capacity and predictable memory access. Designs with similar characteristics include on-DIMM PEs [7], [56] and on-chip PEs within banks [21]. Alternatively, NDAs can utilize high-bandwidth devices, such as the hybrid memory cube (HMC) [65] or high bandwidth memory (HBM) [76]. These offer high internal bandwidth but have limited capacity and high cost due to numerous point-to-point connections to memory controllers [8]. HMC provides capacity scaling via a network but this results in high access latency and cost. HBM does not provide such solutions. As a result, HBM devices are better for standalone accelerators than for main memory.

Operations	Description	Operations	Description
AXPBY	$\vec{z} = \alpha\vec{x} + \beta\vec{y}$	DOT	$c = \vec{x} \cdot \vec{y}$
AXPBYPCZ	$\vec{w} = \alpha\vec{x} + \beta\vec{y} + \gamma\vec{z}$	NRM2	$c = \sqrt{\vec{x} \cdot \vec{x}}$
AXPY	$\vec{y} = \alpha\vec{y} + \vec{x}$	SCAL	$\vec{x} = \alpha\vec{x}$
COPY	$\vec{y} = \vec{x}$	GEMV	$\vec{y} = A\vec{x}$
XMY	$\vec{z} = \vec{x} \odot \vec{y}$		

TABLE I: Example NDA operations used in our case-study application. Chopim is not limited to these operations.

Coherence. Coherence mechanisms between the host and NDAs have been studied in prior NDA work [3], [12], [13] and can be used as is with Chopim. We therefore do not focus on coherence in this paper. In our experiments, we use the existing coherence approach of explicitly and infrequently copying the small amount of data that is not read-only using cache bypassing and memory fences.

Address Translation. Application use of NDAs requires virtual to physical address translation. Some prior work [24], [32], [34] proposes address translation within NDAs to enable independent NDA execution without host assist. This increases both NDA and system complexity. As an alternative, NDA operations can be constrained to only access data within a physical memory region that is contiguous in the virtual address space. Hence, translation is performed by the host when targeting an NDA command at a certain physical address. This has been proposed for both very fine-grain NDA operations within single cache lines [2], [3], [41], [48], [59] and NDA operations within a virtual memory page [61]. In this paper, we use host-based translation because of its low complexity and only check bounds within the NDAs for protection.

NDA Workloads. We focus on NDA workloads for which the host inherently cannot outperform an NDA. These exhibit low temporal locality and low arithmetic intensity and are bottlenecked by peak memory bandwidth. By offloading such operations to the NDA, we mitigate the bandwidth bottleneck by leveraging internal memory module bandwidth. Moreover, these workloads typically require simple logic for computation and integrating such logic within DRAM chips/modules is practical because of the low area and power overhead.

Fundamental linear algebra matrix and vector operations satisfy these criteria. Dense vector and matrix-vector operations, which are prevalent in machine learning primitives, are particularly good candidates because of their deterministic and regular memory access patterns and low arithmetic-intensity. For example, prior work off-loads matrix and vector operations of deep learning workloads to utilize high near-memory BW [25], [40]. Also, Kwon et al. propose to perform element-wise vector reduction operations needed for a deep-learning-based recommendation system to NDAs [45]. In this paper, we focus on accelerating the dense matrix and vector operations summarized in Table I. We demonstrate and evaluate their use in the SVRG application in Section IV. Note that we use these as a concrete example, but our contributions generalize to other NDA operations.

NDA execution of graph processing has also been proposed because graph processing can be bottlenecked by peak memory bandwidth because of low temporal and spatial locality [2], [3], [59], [75], [83]. We do not consider graph processing in this paper because we do not innovate in this context.

III. Chopim

We develop Chopim with four main connected goals that push the state of the art: (1) enable fine-grain interleaving of host and NDA memory requests to the same physical memory devices while mitigating the impact of their contention; (2) permit the use of coarse-grain NDA operations that process long vector instructions/kernels; (3) simultaneously support the locality needed for NDAs and the sophisticated memory address interleaving required for high host performance; and (4) integrate with both a packetized interface and a traditional host-controlled DDRx interface. We detail our solutions in this section after summarizing the need for a new approach.

The need for fine-grain access interleaving with opportunistic NDA issue. An ideal NDA opportunistically issues NDA memory requests whenever a rank is idle from the perspective of the host. This is simple to do in a packetized interface where a memory-side controller schedules all accesses, but is a challenge in a traditional memory interface because the host- and NDA-side controllers must be synchronized. Prior work proposed dedicating some ranks to NDAs and some to the host or coarse-grain temporal interleaving [8], [23]. The former approach contradicts one of our goals as devices are not shared. The latter results in large performance overhead because it cannot effectively utilize periods where a rank is naturally idle due to the host access pattern. Figure 2 shows that for a range of multi-core application mixes (methodology in Section VI), the majority of idle periods are shorter than 100 cycles with the vast majority under 250 cycles. *Fine-grain access interleaving is therefore necessary.*

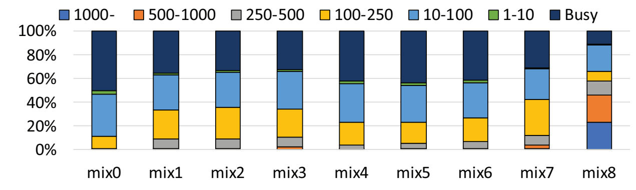


Fig. 2: Rank idle-time breakdown vs. idleness granularity.

The need for coarse-grain NDA vector/kernel operations. Fine-grain access interleaving is simple if each NDA command only addresses a single cache block region of memory. Such fine-grain NDA operations have indeed been discussed in prior work [2], [3], [48], [59]. One overhead of this fine-grain approach is that of issuing numerous NDA commands, with each requiring a full memory transaction that occupies both the command and data channels to memory. Issuing NDA commands too frequently degrades host performance, while infrequent issue underutilizes the NDAs. Coarse-grain NDA vector operations that operate on multiple cache blocks mitigate contention on the channel and improve overall performance. The vector width, N, is specified for each NDA instruction. As long as the operands are contiguous in the DRAM address space, one NDA instruction can process numerous data elements without occupying the channel. Coarse-grain NDA operations are therefore desirable, but *introduce the data layout, memory contention, and host–NDA synchronization challenges which Chopim solves.*

A. Localizing NDA Operands while Distributing Host Accesses

To execute the N-way NDA vector instructions, all the operands of each NDA instruction must be fully contained in a single rank (single PE). If necessary, data is first copied from other ranks prior to launching an NDA instruction. If the reuse rate of the copied data is low, this copying overhead will dominate the NDA execution time and contention on the memory channel will increase due to the copy commands.

We solve this problem in Chopim by laying out data such that all the operands are localized to each NDA at memory allocation time. Thus, copies are not necessary. This is challenging, however, because the host memory controller uses complex address interleaving functions to maximally exploit channel, rank, and bank parallelism for arbitrary host access patterns. Hence, arrays that are contiguous in the host physical address space are not contiguous in physical memory and are shuffled across ranks. This challenge is illustrated in the left side of Figure 3, where two operands of an NDA instructions are shuffled differently across ranks and banks. The layout resulting from our approach is shown at the right of the figure, where arrays (operands) are still shuffled, but both operands follow the same pattern and remain correctly aligned to NDAs without copy operations. Note that alignment is to rank because that corresponds to an NDA partition.

Data layout across ranks.

We rely on the NDA runtime and OS to use a combination of coarse-grain memory allocation and coloring to ensure all operands of an NDA instruction are interleaved across ranks the same way and are thus local to a PE. First, the runtime allocate memory for NDA operands such that they are aligned at the granularity of one DRAM row for each bank in the system which we call a *system row* (e.g., 2MiB for a DDR4 1TiB system). For all the address interleaving mechanisms we are aware of ([53], [67]), this ensures that NDA operands are locally aligned, as long as ranks are also kept aligned. To maintain rank alignment, we reply on OS page coloring to effect rank alignment. We explain this feature below using the Intel Skylake address mapping [67] as a concrete and representative interleaving mapping (Figure 4a).

In this mapping, rank and channel addresses are determined partly by the low-order bits that fall into the frame offset field and partly by the high-order bits that fall into the physical frame number (PFN) field. Frame offsets are kept the same because of the coarse-grain alignment. The OS colors allocations such that the PFN bits that determine rank and channel are aligned for a particular color; which physical address bits select ranks and channels can be reverse engineered if necessary [67]. The Chopim runtime indicates a *shared color* when it requests memory from the OS and specifies the same color for all operands of an instruction. The runtime can use the same color for many operands to minimize copies needed for alignment. In our baseline system, there are 8 colors and each color corresponds to a shared region of memory of 4GiB. Multiple regions can be allocated for the same process. Though we focus on one address mapping here, our approach works with any linear address mapping described in prior work [53], [67] as well.

Note that coarse-grain allocation is simple with the common buddy allocator if allocation granularity is also a system row, and can use optimizations that already exist for huge pages [27], [46], [81]. The fragmentation overheads of coarse allocation are similar to those with huge pages and we find that they are negligible because coarse-grain NDA execution works best when processing long vectors.

Data layout across DRAM chips. In the baseline system, each 4-byte word is striped across multiple chips, whereas in our approach each word is located in a single chip so that NDAs can access words from their local memory. Both the host and NDAs can access memory without copying or reformatting data (as required by prior work [23]). Memory blocks still align with cache lines, so this layout change is not visible to software. Note that this data layout does not impact the host memory controller's ECC computation (e.g. Chip-kill [20]) because ECC protects only bits and not how they are interpreted. For NDA accesses, we rely on in-DRAM ECC with its limited coverage. We do not innovate in this respect and leave this problem for future work.

B. Mitigating Frequent Read/Write Penalties

The basic memory access scheduling policy we use for Chopim is to always prioritize host memory requests, yet aggressively leverage unutilized rank bandwidth by issuing NDA requests whenever possible. That is, NDAs wait when incoming host requests are detected, but otherwise always issue their memory requests to maximize their bandwidth utilization and performance. One potential problem is that an NDA request issued in one cycle may delay a host request that could have issued in one of the following cycles otherwise.

We find that NDAs infrequently issue row commands (ACT and PRE). We therefore prioritize host memory commands

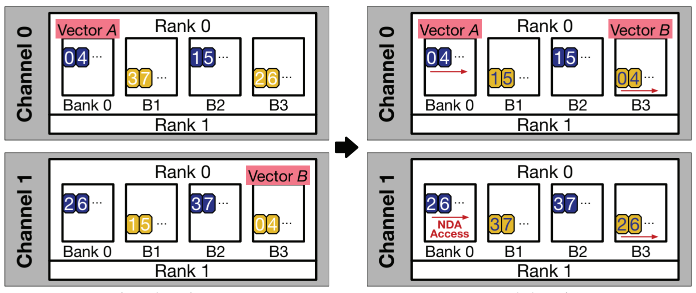
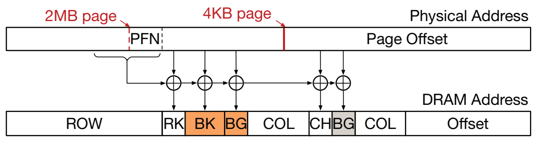
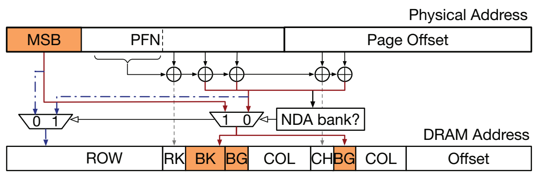


Fig. 3: Example data layout across ranks for concurrent access of the COPY operation (B[i] = A[i]). With naive data layout (left), elements with the same index are located in different ranks. With our proposed mechanism (right), elements with the same index are co-located. NDAs access contiguous columns starting from the base of each vector.



(a) Baseline (Skylake [67])



(b) Proposed (for bank partitioning)

Fig. 4: Baseline and proposed host-side address mapping.

over any NDA row command to the same bank. This has negligible impact on NDA performance in our experiments.

We also find that read transactions of NDAs have only a small impact on following host commands. NDA write transactions, however, can have a large impact on host performance because of the read/write-turnaround penalties that they frequently require. While the host mitigates turnaround overhead by buffering operations with caches and write buffers [1], [78], the host and NDAs may interleave different types of transactions when accessing memory in parallel. We find that NDA writes interleaved with host reads degrade performance the most. *As a solution,* we introduce two mechanisms to selectively throttle NDA writes.

Our first mechanism throttles the rate of NDA writes by issuing them with a predefined probability. We call this mechanism *stochastic NDA issue*. Before issuing a write transaction, the NDAs both detect if a rank is idle and flip a coin to determine whether to issue the write. By adjusting the coin weight, the performance of the host and NDAs can be traded off: higher write-issue probability leads to more frequent turnarounds while a lower probability throttles NDA progress. Deciding how much to throttle NDAs requires analysis or profiling, and we therefore propose a second approach as well.

Our second approach does not require tuning, and we empirically find that it works well. In this *next rank prediction* approach, the memory controller inhibits NDA write requests when more host read requests are expected; the controller stalls the NDA in lieu of providing an NDA write queue. In a packetized interface, the memory controller schedules both host and NDA requests and is thus aware of potential required turnarounds. The traditional memory interface, however, is more challenging as the host controller must explicitly signal the NDA controller to inhibit its write request. This signal must be sent ahead of the regular host transaction because of bus delays.

We use a very simple predictor that inhibits NDA write requests in a particular rank when the oldest outstanding host memory request to that channel is a read to that same rank. Specifically, the NDA controller examines the target rank of the oldest request in the host memory controller transaction queue. Then, it signals to the NDAs in that rank to stall their writes. For now, we assume that this information is communicated over a dedicated pin and plan to develop other signaling mechanisms that can piggyback on existing host DRAM commands at a later time. Our experiments with an FRFCFS [70] memory scheduler at the host show that this simple predictor works well and achieves performance that is comparable to a tuned stochastic issue approach.

C. Partitioning into Host and Shared Banks

In addition to read/write-turnaround overheads, concurrent access also degrades performance by decreasing DRAM row access locality. When the host and NDAs interleave accesses to different rows of the same bank, frequent bank conflicts occur. To avoid this bank contention, we propose using bank partitioning to limit bank interference to only those memory regions that must concurrently share data between the NDAs and the host. This is particularly useful in colocation scenarios when only a small subset of host tasks utilize the NDAs. However, existing bank partitioning mechanisms [36], [52], [57] are incompatible with both huge pages and with sophisticated DRAM address interleaving schemes.

Bank partitioning relies on the OS to color pages where colors can be assigned to different cores or threads, or in our case, for banks isolated for the host and those that could be shared. The OS then maps pages of different color to frames that map to different banks. Figure 4a shows an example of a modern physical address to DRAM address mapping [67]. One color bit in the baseline mapping belongs to the page offset field so prior bank partitioning schemes can, at best, be done at two-bank granularity. More importantly, when huge pages are used (e.g., 2MiB), this baseline mapping cannot be used to partition banks at all.

To overcome this limitation, we propose a new interface that partitions banks into two groups—host-reserved and shared banks—with flexible DRAM address mapping and any page size. Specifically, our mechanism only requires that the most significant physical address bits are only used to determine DRAM row address, as is common in recent hash mapping functions, as shown in Figure 4b [67].

Without loss of generality, assume 2 banks out of 16 banks are reserved for the shared data. First, the OS splits the physical address space for host-only and shared memory region with the host-only region occupying the bottom of the address space: $0 - (14 \times (bank\_capacity) - 1)$. The rest of the space (with the capacity of 2 banks) is reserved for the shared data and the OS does not use it for other purposes. This guarantees that the most significant bits (MSBs) of the address of host-only region are never b'111. In contrast, addresses in the shared space always have b'111 in their MSBs.

The OS informs the memory controller that it reserved 2 banks (the top-most banks) for shared memory region. Host-only memory addresses are mapped to DRAM locations using any hardware mapping function, which is not exposed to software and the OS. The idea is then to remap addresses that initially fall into shared banks into the reserved address space that the host is not using. Additional simple logic checks whether the resulting DRAM address bank ID of the initial mapping is a reserved bank for shared region. If they are not, the DRAM address is used as is. If the DRAM address is initially mapped to one of the reserved banks, the MSBs and the bank bits are swapped. Because the MSBs of a host address are never b'1110 or b'1111, the final bank ID will be one of the host-only bank IDs. Also, because the bank ID of the initial mapping result is 14 or 15, the final address is in a row the host cannot access with the initial mapping and there is no aliasing. Note that the partitioning decision can be adjusted, but only if all affected memory is first cleared.

D. Tracking Global Memory Controller State

Unlike conventional systems, Chopim also enables an architecture that has two memory controllers (MCs) managing the bank and timing state of each rank. This is the case when the host continues to directly manage memory even when the memory itself is enhanced with NDAs. This requires coordinating rank state information between controllers. Figure 5 shows how MCs on both sides of a memory channel track global memory controller state. Information about host transactions is easily obtained by the NDA MCs as they can monitor incoming transactions and update the state tables accordingly (left). However, the host MC cannot track all NDA transactions due to command bandwidth limits.

To solve this problem, we replicate the finite-state machines (FSMs) of NDAs and place them in the host-side NDA controller. When an NDA instruction is launched, the FSMs on both sides are synchronized. We rely on the already-synchronized DDR interface clock for FSM synchronization. Whenever an NDA memory transaction is issued, the host-side FSM also updates the state table in the host MC without communicating with the NDAs (right). If a host transaction blocks NDA transactions in one of the ranks, that transaction will be visible to both FSMs. Replicated FSMs track the NDA write buffer occupancy and detect when the write-buffer draining starts and ends to trigger write throttling. The area and

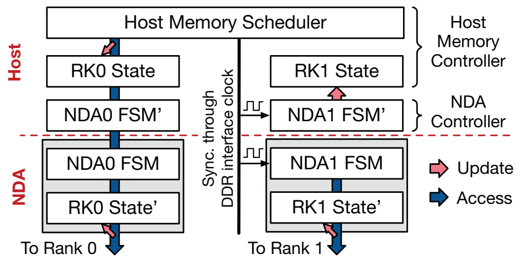
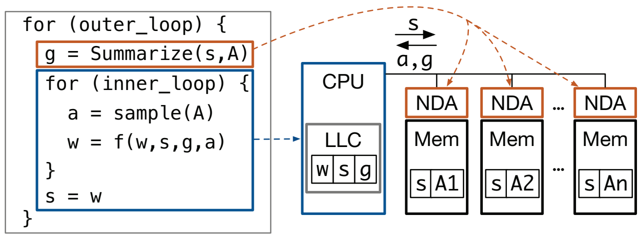


Fig. 5: Global MC state tracking when the host (left) and NDAs (right) issues memory commands. The replicated FSMs are synchronized by using the DDR interface clock.



```
for (outer_loop) {
  g = Summarize(s,A)
  for (inner_loop) {
    a = sample(A)
    w = f(w,s,g,a)
  }
  s = w
}
```

Fig. 6: Collaboration between host and NDAs in SVRG.

power overhead of replicating FSMs are negligible (40-byte microcode store and 20-byte state registers per rank (i.e., per NDA)). *Our evaluation uses this approach to enable a DDR4-based NDA-enabled main memory and all our experiments rely on this.*

IV. Host-NDA Collaboration

In this section, we describe a case study to show the potential of concurrent host-NDA execution by collaboratively processing the same data. Our case study shows how to partition an ML training task between the host and NDAs such that each processor leverages its strengths. As is common to training and many data-processing tasks, the vast majority of shared data is read-only, simplifying parallelism.

We use the machine-learning technique of logistic regression with stochastic variance reduced gradient (SVRG) [37] as our case study. Figure 6 shows a simplified version of SVRG and the opportunity for collaboration. The algorithm consists of two main tasks within each outer-loop iteration. First, the entire large input matrix A is *summarized* into a single vector g (see Figure 8 for pseudocode). This vector is used as a correction term when updating the model in the second task. This second task consists of multiple inner-loop iterations. In each inner-loop iteration the learned model w based on a randomly-sampled vector a from the large input matrix A, the correction term g, and a stored model s, which is updated at the end of the outer-loop iteration.

The first task is an excellent match for the NDAs. The summarization operation is simple, exhibits little reuse, and traverses the entire large input data. In contrast, the second task with its tight inner loop is well suited for the host. The host can maximally exploit locality captured by its caches while NDAs can leverage their high bandwidth for accessing the entire input data A. Note that in SVRG, an *epoch* refers to the number of inner loop iterations.

The main tradeoff in SVRG is as follows. When summarization is done more frequently, the quality of the correction term increases and, consequently, the per-step convergence rate increases. On the other hand, the overhead of summarization also increases when it is performed more frequently, which offsets the improved convergence rate. Therefore, the *epoch* hyper-parameter, which determines the frequency of summarization, should be carefully selected to optimize this tradeoff.

Delayed-Update SVRG. As Chopim enables concurrent access from the host and NDAs, we explore an algorithm change to leverage collaborative parallel processing. Instead of alternating between the summarization and model update tasks, we run them in parallel on the host and NDAs. Whenever the NDAs finish computing the correction term, the host and NDAs exchange the correction term and the most up-to-date weights before continuing parallel execution. While parallel execution is faster, it results in using stale s and g values from one epoch behind. The main tradeoff in *delayed-update SVRG* is that per-iteration time is improved by overlapping execution, whereas convergence rate per iteration degrades due to the staleness. Similar tradeoffs have been observed in prior work [9], [19], [47], [69]. We later show that delayed-update SVRG can converge in 40% less time than when serializing the two main SVRG tasks.

To avoid races for s and g in this delayed-update SVRG, we maintain private copies of these small variables and use a memory fence that guarantees completion of DRAM writes after the data-exchange step (which the runtime coordinates with polling). Note that we bypass caches when accessing data produced/consumed by NDAs during the data-exchange step. Since s and g are small and copied infrequently, the overheads are small and amortized over numerous NDA computations. Whether delayed updates are used or not, the host and NDAs share the large data, A, without copies.

V. Runtime and API

Chopim is general and helps whenever host/NDA concurrent access is needed. To make the explanations and evaluation concrete, we use an exemplary interface design as discussed below and summarized in Figure 7. Command and address signals pass through the NDA memory controllers so that they can track host rank state. Processing elements (PEs) in the logic die access data by using their local NDA memory controller (Figure 1). Figure 8 shows example usage of our API for computing the average gradient used in the summarization task of SVRG.

The Chopim runtime system manages memory allocations and launches NDA operations. NDA operations are blocking by default, but can also execute asynchronously. If the programmer calls an NDA operation with operands from different shared regions (colors), the runtime system inserts appropriate data copies. We envision a just-in-time compiler that can

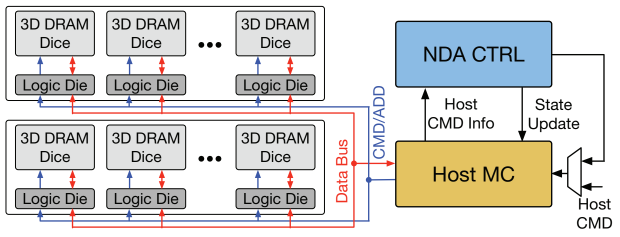


Fig. 7: Overview of NDA architecture.

identify such cases and more intelligently allocate memory and regions to minimize copies. For this paper, we do not implement such a compiler. Instead, programs are written to directly interact with a runtime system that is implemented within the simulator.

NDAs operate directly on DRAM addresses and do not perform address translation. To launch an operation, the runtime (with help from the OS) translates the origin of each operand into a physical address, which is then communicated along with a bound to the NDAs by the NDA controller. The runtime is responsible for splitting a single API call into multiple primitive NDA operations. The NDA operations themselves proceed through each operand with a regular access pattern implemented as microcode in the hardware, which also checks the bound for protection.

Optimization for Load-Imbalance. Load imbalance occurs when the host does not access ranks uniformly over short periods of time. The AXPY operation (launched repeatedly within the loop shown in Figure 8) is short and non-uniform access by the host leads to load imbalance among NDAs. A blocking operation waits for *all* NDAs to complete before launching the next AXPY, which reduces performance. Our API provides asynchronous launches similar to CUDA streams or OpenMP `parallel for` with a `nowait` clause [18]. Asynchronous launches can overlap AXPY operations from multiple loop iterations. Any load imbalance is then only apparent when the loop ends. Over such a long time period, load imbalance is much less likely. We implement asynchronous launches using *macro NDA operation*. An example of a macro operation is shown in the loop of Figure 8 and is indicated by the `parallel_for` annotation.

Launching NDA Operations. NDA operations are launched similarly to Farmahini et al. [23]. A memory region is reserved for accessing control registers of NDAs. NDA packets access the control registers and launch operations. Each packet is composed of the type of operation, the base addresses of operands, the size of data blocks, and scalar values required for scalar-vector operations. On the host side, the *NDA controller* plays two main roles. First, it accepts acceleration requests, issues commands to the NDAs in the different ranks (in a round-robin manner), and notifies software when a request completes. Second, it extends the host memory controller to coordinate actions between the NDAs and host memory controllers and enables concurrent access. It maintains the replicated FSMs using its knowledge of issued NDA operations and the status

```
void main () {
  // Memory Allocation
  float alpha, lambda;
  nda::matrix<float> X(n, d, nda::SHARED);
  nda::vector<float> w(d, nda::SHARED);
  nda::vector<float> y(n, nda::SHARED);
  nda::vector<float> v(n, nda::SHARED);
  nda::vector<float> a(d, nda::SHARED);
  nda::vector<float> a_pvt(d, nda::PRIVATE);
  // a_pvt: allocates d elements per NDA rather than
  //        striping the allocation across NDAs

  // Initialization
  ...

  // Average Gradient
  nda::gemv(y, X, w);
  nda::xmy(v, v, y);
  host::sigmoid(v, v);
  nda::xmy(v, v, y);
  nda::scal(v, 1/n);

  // Target for Macro Operation
  parallel_for (int i = 0; i < n; i++) {
    alpha = v[i];
    nda::axpy(a_pvt, alpha, X, i); // a_pvt += alpha *X[i]
  }

  host::reduce(a, a_pvt);
  nda::axpy(a, lambda, w);
}
```

Fig. 8: Average gradient example code. This code corresponds to *summarization* in SVRG (see Section IV).

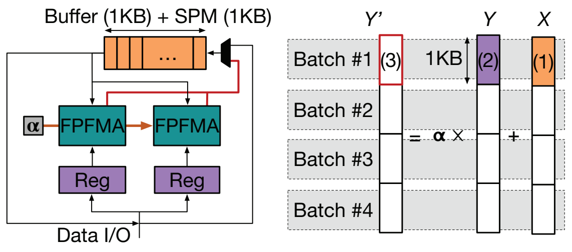


Fig. 9: PE architecture and execution flow of AXPY.

of the host memory controller.

Execution Flow of a Processing Element. Our exemplary PE is composed of two floating-point fused multiply-add (FPFMA) units, 5 scalar registers (up to 3 operand inputs and 2 for temporary values), a 1KB buffer for accessing memory, and the 1KB scratchpad memory. The memory access granularity is 8B per chip and the performance of the two FPFMAs per chip matches this data access rate. PEs may be further optimized to support lower-precision operations or specialized for specific use cases, but we do not explore these in this paper as we focus on the new capabilities of Chopim rather than NDA in general.

Figure 9 shows the execution flow of a PE when executing the AXPY operation. Each vector is partitioned into 1KB batches, which is the same size as DRAM page size per chip. To maximize bandwidth utilization, the vector X is streamed into the buffer. Then, the PE opens another row, reads two elements (8 bytes) of vector Y, and stores them to FP registers. While the next two elements of Y are read, a fused multiply-add (FMA) operation is executed. The result is stored back into the buffer and execution continues such that the read-execute-write operations are pipelined. After the result buffer is filled, the PE either writes results back to memory or to the scratchpad. This flow for one 1KB batch is repeated over the rest of the batches. This entire process is stored in PE microcode as the AXPY operation. Other operations (coarse or fine grained) are similarly stored and processed from microcode.

Inter-PE Communication. NDAs are only effective when they use memory-side bandwidth to amplify that of the host. In the DIMM- and chip-based NDAs, which we target in this paper, general inter-PE communication is therefore equivalent to communicating with the host. Communication in applications that match this NDA architecture are primarily needed for replicating data to localize operands or for global reduction operations, which follow local per-PE reductions. In both communication cases, a global view of data layout is needed and, therefore, we enable communication only through the host. For instance, after the macro operation in Figure 8, a global reduction of the PE private copies (`a_pvt`) accumulates the data for the final result (`a`). The reduced result is used by the following NDA operation, requiring replication communication for its data layout has to meet NDA locality requirements with the other NDA operand (`w`). Though communicating through the host is expensive, our coarse-grained NDA operations amortize infrequent communication overhead. Importantly, since this communication can be done as normal DRAM accesses by the host, no change on the memory interface is required.

System configuration	
Processor	4-core OoO x86 (8 cores for mix0), 4GHz, Fetch/Issue width (8), LSQ (64), ROB (224)
NDA	one PE per chip, 1.2GHz, fully pipelined, write buffer (128) (Section V)
TLB	I-TLB:64, D-TLB:64, Associativity (4)
L1	32KB, Associativity (L1I: 8, L1D: 8), LRU, 12 MSHRs
L2	256KB, Associativity (4), LRU, 12 MSHRs
LLC	8MB, Associativity (16), LRU, 48 MSHRs, Stride prefetcher
DRAM	DDR4, 1.2GHz, 8Gb, x8, 2channels $\times$ 2ranks, FR-FCFS, 32-entry RD/WR queue, Open policy, Intel Skylake address mapping [67]

DRAM timing parameters
tBL=4, tCCDS=4, tCCDL=6, tRTRS=2, tCL=16, tRCD=16, tRP=16, tCWL=12, tRAS=39, tRC=55, tRTP=9, tWTRS=3, tWTRL=9, tWR=18, tRRDS=4, tRRDL=6, tFAW=26

Energy Components
Activate energy: 1.0nJ, PE read/write energy: 11.3pJ/b, host read/write energy: 25.7pJ/b, PE FMA: 20pJ/operation, PE buffer dynamic: 20pJ/access, PE buffer leakage power: 11mW (Energy/power of scratchpad memory is same as PE buffer)

Benchmarks		MPKI
mix0	mcf_r:lbm_r:omnetpp_r:gemsFDTD bwaves:milc:soplex:leslie3d	H:H:H:H H:M:M:M
mix1	mcf_r:lbm_r:omnetpp_r:gemsFDTD	H:H:H:H
mix2	mcf_r:lbm_r:gemsFDTD:soplex	H:H:H:H
mix3	lbm_r:omnetpp_r:gemsFDTD:soplex	H:H:H:H
mix4	omnetpp_r:gemsFDTD:soplex:milc	H:H:H:M
mix5	gemsFDTD:soplex:milc:bwaves_r	H:H:M:M
mix6	soplex:milc:bwaves_r:leslie3d	H:M:M:M
mix7	milc:bwaves_r:astar:cactusBSSN_r	M:M:M:M
mix8	leslie3d:leela_r:deepsjeng_r:xchange2_r	M:L:L:L

NDA Kernels
NDA basic operations (Table I), SVRG (details below), CG (16K $\times$ 16K), and SC (2M $\times$ 128)

Machine Learning Configurations
Logistic regression with ℓ2-regularization (10-class classification), λ=1e-3, learning rate=best-tuned, momentum=0.9, dataset=cifar10 (50000 $\times$ 3072)

TABLE II: Evaluation parameters.

VI. Methodology

Table II summarizes our system configuration, DRAM timing parameters, energy components, benchmarks, and machine learning configurations. For bank partitioning, we reserve one bank per rank for NDAs and the rest for the host. We use Ramulator [42] as our baseline DRAM simulator and add the NDA memory controllers and PEs to execute the NDA operations. We modify the memory controller to support the Skylake address mapping [67] and our bank partitioning and data layout schemes. To simulate concurrent host accesses, we use gem5 [10] with Ramulator. We choose host applications that have various memory intensity from the *SPEC2006* [31] and *SPEC2017* [62] benchmark suites and form 9 different application mixes with different combinations (Table II). Mix0 and mix8 represent two extreme cases with the highest and lowest memory intensity, respectively. Only mix0 is run with 8 cores to simulate under-provisioned bandwidth while other mixes use 4 cores to simulate a more realistic scenario. For the NDA workloads, we use DOT and COPY operations to show the impact of extremely low and high write intensity. We use the average gradient kernel (Figure 8) to evaluate collaborative execution. The performance impact of other NDA applications falls between DOT and COPY and is well represented by SVRG [37], conjugate gradient (CG) [35] and streamcluster (SC) [68].

For the host workloads, we use Simpoint [30] to find representative program phases and run each simulation until the instruction count of the slowest process reaches 200M instructions. If an NDA workload completes while the simulation is still running, it is relaunched so that concurrent access occurs throughout the simulation time. Since the number of instructions simulated is different, we measure instructions per cycle (IPC) for the host performance. To show how well the NDAs utilize bandwidth, we show bandwidth utilization and compare with the idealized case where NDAs can utilize all the idle rank bandwidth.

We estimate power with the parameters in Table II. We use CACTI 6.5 [58] for the dynamic and leakage power of the PE buffer. A sensitivity study for PE parameters exhibits that their impact is negligible. We use CACTI-3DD [15] to estimate the power and energy of 3D-stacked DRAM and CACTI-IO [38] to estimate DIMM power and energy.

VII. Evaluation

We present evaluation results for the various Chopim mechanisms, analyzing: (1) the benefit of coarse-grain NDA operations; (2) how bank partitioning improves NDA performance; (3) how stochastic issue and next-rank prediction mitigate read/write turnarounds; (4) the impact of NDA workload write intensity and load imbalance; (5) how Chopim compares with rank partitioning; (6) the benefits of collaborative and parallel

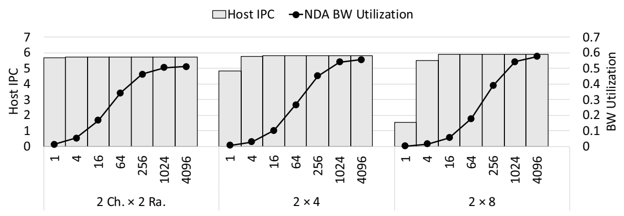


Fig. 10: Impact of coarse-grain NDA operations. (X-axis: the number of cache blocks accessed per NDA instruction.)

CPU/NDA processing; and (7) energy efficiency. All results rely on the replicated FSM to enable using DDR4.

Coarse-grain NDA Operation. Figure 10 demonstrates how overhead for launching NDA instructions can degrade performance of the host and NDAs as rank count increases. To prevent other factors, such as bank conflicts, bank-level parallelism, and load imbalance from affecting performance, we use our BP mechanism, the NRM2 operation (because we can precisely control its granularity), and asynchronous launch. We run the most memory-intensive application mix (mix1) on the host. When more CBs are processed by each NDA instruction, contention between host transactions and NDA instruction launches decreases and performance of both improves. In addition, as the number of ranks grows, contention becomes severe because more NDA instructions are necessary to keep all NDAs busy. These results show that our data layout that enables coarse-grain NDA operation is beneficial, especially in concurrent access situation.

Takeaway 1: Coarse-grain NDA operations are crucial for mitigating contention on the host memory channel.

Impact of Bank Partitioning. Figure 11 shows performance when banks are shared or partitioned between the host and NDAs. We emphasize the impact of write intensity of NDA operations by running the extreme DOT (read intensive) and COPY (write intensive) operations. While not shown, SVRG falls roughly in the middle of this range. We compare each memory access mode with an idealized case where we assume the host accesses memory without any contention and NDAs can leverage all the idle rank bandwidth without considering transaction types and other overheads.

Overall, accelerating the read-intensive DOT with concurrent host access does not affect host performance significantly even with our aggressive approach. However, contention with the shared access mode significantly degrades NDA performance. This is because of the extra bank conflicts caused by interleaving host and NDA transactions. On the other hand, accelerating the write-intensive COPY degrades host performance. This happens because, in the write phase of NDAs when the NDA write buffer drains, the host reads are blocked while NDAs keep issuing write transactions due to long write-to-read turnaround time. To mitigate this problem, we show the impact of our write throttling mechanisms below. Note that host performance of mix0 is the lowest, despite its

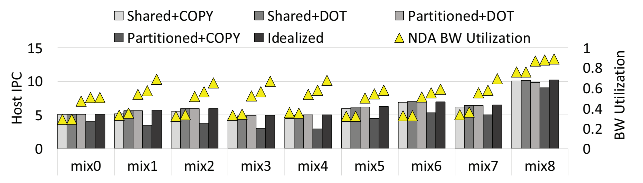


Fig. 11: Concurrent access to different memory regions.

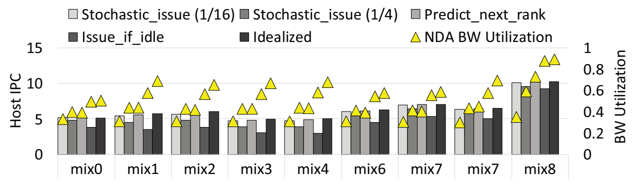


Fig. 12: Stochastic issue and next-rank prediction impact.

doubled core count, because contention for LLC increases and memory performance dominates overall performance.

Takeaway 2: Bank partitioning increases row-buffer locality and substantially improves NDA performance, especially for read-intensive NDA operations.

Mitigating NDA Write Interference. Figure 12 shows the impact of mechanisms for write-intensive NDA operations. In this experiment, the most write-intensive operation, COPY, is executed by NDAs and the mechanisms are applied only during the write phase of NDA execution. Stochastic issue is used with two probabilities, 1/4 and 1/16, which clearly shows the host-NDA performance tradeoff compared to next-rank prediction.

For stochastic issue, the tradeoff between host and NDA performance is clear. If NDAs issue with high probability, host performance degrades. The appropriate issue probability can be chosen with heuristics based on host memory intensity though we do not explore this in this paper. On the other hand, the next-rank prediction mechanism shows slightly better behavior than the stochastic approach. Compared to stochastic issue with probability 1/16, both host and NDA performance are higher. Stochastic issue extends the tradeoff range and does not require signaling. We use the robust next-rank prediction approach for the rest of the paper.

Takeaway 3: Throttling NDA writes mitigates the large impact of read/write turnaround interference on host performance; next-rank prediction is robust and effective while stochastic issue does not require additional signaling.

Impact of Write-Intensity and Input Size. Figure 13 shows host and NDA performance when different types of NDA operations are executed with different input sizes. The host application mix with the highest memory intensity (mix1) and the next-rank prediction mechanism is used. In addition, to identify the impact of input size, three different vector sizes are used: small (8KB/rank), medium (128KB/rank), and large (8MB/rank). We evaluate asynchronous launches with the small vector size. We evaluate GEMV with three matrix

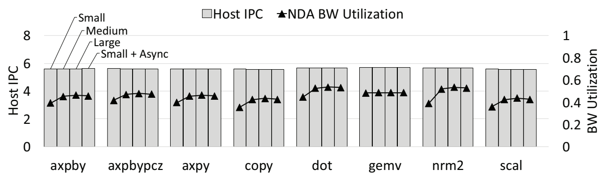


Fig. 13: Impact of NDA operations and operand size.

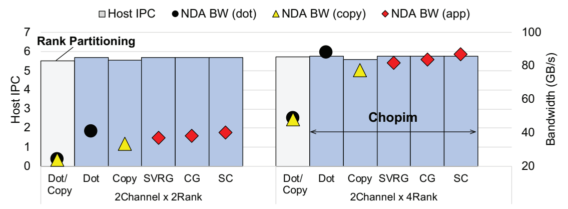


Fig. 14: Scalability Chopim vs. rank partitioning.

sizes, where the number of columns is equal to each of the three vector sizes and the number of rows fixed at 128.

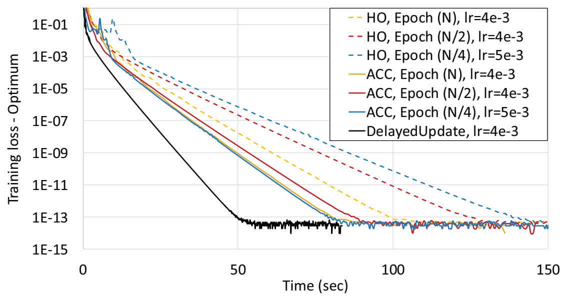
Overall, performance is inversely related to write intensity, and short execution time per launch results in low NDA performance. The NRM2 operation with the small input has the shortest execution time. Because of its short execution time, NRM2 is highly impacted by the launching overhead and load imbalance caused by concurrent host access. On the other hand, GEMV executes longer than other operations and it is impacted less by load imbalance and launching overhead. With the asynchronous launch optimization, the impact of load imbalance decreases and NDA bandwidth increases.

Takeaway 4: Asynchronous launch mitigates the load imbalance caused by short-duration NDA operations.

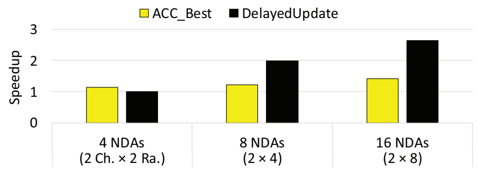
Scalability Comparison. Figure 14 compares Chopim with the performance of rank partitioning (RP). For RP, we assume that ranks are evenly partitioned between the host and NDAs. Since read- and write-intensive NDA operations show different trends, we separate those two cases. Other application results (SVRG, CG, and SC) are shown to demonstrate that their performance falls between these two extreme cases. We use the most memory-intensive mix1 as the host workload. The first cluster shows performance when the baseline DRAM system is used. For both the read- and write-intensive NDA workloads, Chopim performs better than rank partitioning. This shows that opportunistically exploiting idle rank bandwidth can be a better option than dedicating ranks for acceleration. The second cluster shows performance when the number of ranks is doubled. Compared to rank partitioning, Chopim shows better performance scalability. While NDA bandwidth with rank partitioning exactly doubles, Chopim more than doubles due to the increased idle time per rank. SVRG results fall between extreme DOT and COPY cases.

Takeaway 5: Chopim scales better than rank partitioning because short issue opportunities grow with rank count.

SVRG Collaboration Benefits. Figure 15a shows the convergence results with and without NDA (8 NDAs). We use



(a) Convergence over time with and without NDA.



(b) NDA speedup scaling (normalized to host only).

Fig. 15: Impact of NDA summarization in SVRG with and without delayed update (HO: Host-Only, ACC: Accelerated with NDAs, ACC_Best: Best among all ACC options).

a shared memory region to enable concurrent access to the same data and the next-rank prediction mechanism is used. Compared to the host-only case, the optimal epoch size decreases from *N* to *N/4* when NDAs are used. This is because the overhead of summarization decreases relative to the host-only case. Furthermore, SVRG with delayed updates gains additional performance demonstrating the benefits made possible by the concurrent host and NDA access when each processes the portion of the workload it is best suited for. Though the delayed update updates the correction term more frequently, the best performing learning rate is lower than ACC with epoch *N/4*, which shows the impact of staleness on the delayed update.

When NDA performance grows by adding NDAs (additional ranks), delayed-update SVRG demonstrates better performance scalability. Figure 15b compares the performance of the best-tuned serialized and delayed-update SVRG with that of host-only with different number of NDAs. We measure performance as the time it takes the training loss to converge (when it reaches $1e-13$ away from optimum). Because more NDAs can calculate the correction term faster, its staleness decreases, consequently, a higher learning rate with faster convergence is possible.

Takeaway 6: Collaborative host-NDA processing on shared data speeds up SVRG logistic regression by 50%.

Memory Power. We estimate the power dissipation in the memory system under concurrent access. The theoretical maximum possible power of the memory system is 8W when only the host accesses memory. When the most memory-intensive application mixes are executed, the average power is 3.6W. The maximum power of NDAs is 3.7W and is dissipated when the scratchpad memory is maximally used in the average gradient computation. In total, up to 7.3W of power is dissipated in the memory system, which is lower than the maximum possible with host-only access. This power efficiency of NDAs comes from the low-energy internal memory accesses and because Chopim minimizes overheads.

Takeaway 7: Operating multiple ranks for concurrent access does not increase memory power significantly.

VIII. Related Work

To the best of our knowledge, this is the first work that proposes solutions for near data acceleration while enabling the concurrent host and NDA access without data reorganization and in a non-packetized DRAM context. Packetized DRAM, while scalable, may suffer from 2–4x latency longer than DDR-based protocol even under very low or no load [29]. To solve this unique problem, many previous studies have influenced our work.

The study of near data acceleration has been conducted in a wide range as the relative cost of data access becomes more and more expensive compared to the computation itself. The nearest place for computation is in DRAM cells [49], [71], [72] or the crossbar cells with emerging technologies [14], [16], [50], [55], [73]–[75], [79]. Since the benefit of near-data acceleration comes from high bandwidth and low data transfer energy, the benefit becomes larger as computation move closer to memory. However, area and power constraints are significant, restricting adding complex logic. As a result, workloads with simple ALU operations are the main target of these studies.

3D stacked memory devices enable more complex logic on the logic die and still exploit high internal memory bandwidth. Many recent studies are conducted based on this device to accelerate diverse applications [2], [3], [12], [13], [22], [24], [25], [28], [32]–[34], [40], [51], [54], [60], [64], [82]. However, in these proposals, the main memory role of the memory devices has gained less attention compared to the acceleration part. Some prior work [4], [5], [11], [80] attempts to support the host and NDA access to the same data but only with data reorganization and in a packetized DRAM context. Parrnaik et al. [64] show the potential of concurrently running both the host and NDAs on the same memory. However, they assume an idealized memory system in which there is no contention between NDA and host memory requests. We do not assume this ideal case. The main contributions of Chopim are precisely to provide mechanisms for mitigating interference.

On the other hand, *NDA* [23], Chameleon [8], and MCN DIMM [7] are based on conventional DIMM devices and changes the DRAM design to practically add PEs. Unlike rank partitioning and coarse-grain mode switching used in the prior work, we let host and PEs share ranks to maximize parallelism and partition banks to decrease contention.

IX. Conclusion

In this paper, we introduced solutions to share ranks and enable concurrent access between the host and NDAs. Instead of partitioning memory in coarse-grain manner, both temporally and spatially, we interleave accesses in fine-grain manner to leverage the unutilized rank bandwidth. To maximize bandwidth utilization, Chopim enables coordinating state between the memory controllers of the host and NDAs in low overhead, to reduce extra bank conflicts with bank partitioning, to efficiently block NDA write transactions with stochastic issue and next-rank prediction to mitigate the penalty of read/write turnaround time, and to have one data layout that allows the host and NDAs to access the same data and realize high performance. Our case study also shows that collaborative execution between the host and NDAs can provide better performance than using just one of them at a time. Chopim offers insights to practically enable NDA while serving main memory requests in real systems and enables more effective acceleration by eliminating data copies and encouraging tighter host-NDA collaboration.

References

[1] Jung Ho Ahn, Mattan Erez, and William J Dally. The design space of data-parallel memory systems. In *SC06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, June 2015.

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

[4] Berkin Akin, Franz Franchetti, and James C. Hoe. Data reorganization in memory using 3d-stacked dram. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 131–143, New York, NY, USA, 2015. ACM.

[5] Berkin Akin, Franz Franchetti, and James C Hoe. Hamlet architecture for parallel data reorganization in memory. *IEEE Micro*, 36(1):14–23, Jan 2016.

[6] Mohammad Alian and Nam Sung Kim. Netdimm: Low-latency near-memory network interface architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 699–711. ACM, 2019.

[7] Mohammad Alian, Seung Won Min, Hadi Asgharimoghaddam, Ashutosh Dhar, Dong Wang, Adam Roewer, Thomas McPadden, Oliver OHalloran, Deming Chen, Jinjun Xiong, Daehoon Kim, Wen-mei Hwu, and Nam Sung Kim. Application-transparent near-memory processing architecture with memory channel network,. In *The 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

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

[9] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[11] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur 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*, ASPLOS '18, pages 316–331, New York, NY, USA, 2018. ACM.

[12] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. Conda: Efficient cache coherence support for near-data accelerators. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 629–642, New York, NY, USA, 2019. ACM.

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

[14] Fan Chen, Linghao Song, and Yiran Chen. Regan: A pipelined reram-based accelerator for generative adversarial networks. In *Design Automation Conference (ASP-DAC), 2018 23rd Asia and South Pacific*, pages 178–183. IEEE, 2018.

[15] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 33–38. IEEE, 2012.

[16] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 27–39. IEEE Press, 2016.

[17] JS Choi. Next big thing: Ddr4 3ds.

[18] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *Computing in Science & Engineering*, 5(1):46–55, 1998.

[19] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[20] Timothy J Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory. *IBM Microelectronics Division*, 11:1–23, 1997.

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

[22] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. The mondrian data engine. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 639–651. ACM, 2017.

[23] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 283–295. IEEE, 2015.

[24] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 113–124. IEEE, 2015.

[25] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–764. ACM, 2017.

[26] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The terasys massively parallel pim array. *Computer*, 28(4):23–31, 1995.

[27] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[28] Qi Guo, Nikolaos Alachiotis, Berkin Akin, Fazle Sadi, Guanglin Xu, Tze Meng Low, Larry Pileggi, James C Hoe, and Franz Franchetti. 3d-stacked memory-side acceleration: Accelerator and system design. In *In the Workshop on Near-Data Processing (WoNDP)(Held in conjunction with MICRO-47.)*, 2014.

[29] Ramyad Hadidi, Bahar Asgari, Jeffrey Young, Burhan Ahmad Mudassar, Kartikay Garg, Tushar Krishna, and Hyesoon Kim. Performance implications of nocs on 3d-stacked memories: Insights from the hybrid memory cube. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 99–108. IEEE, 2018.

[30] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

[31] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[32] Byungchul Hong, Gwangsun Kim, Jung Ho Ahn, Yongkee Kwon, Hongsik Kim, and John Kim. Accelerating linked-list traversal through near-data processing. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, 2016.

[33] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 204–216, June 2016.

[34] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 25–32. IEEE, 2016.

[35] B Jacob, G Guennebaud, et al. Eigen: C++ template library for linear algebra, 2013.

[36] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Mike Sullivan, Ikhwan Lee, and Mattan Erez. Balancing dram locality and parallelism in shared memory cmp systems. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[37] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems*, pages 315–323, 2013.

[38] Norman P Jouppi, Andrew B Kahng, Naveen Muralimanohar, and Vaishnav Srinivas. Cacti-io: Cacti with off-chip power-area-timing models. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(7):1254–1267, 2015.

[39] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. Flexram: Toward an advanced intelligent memory system. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040)*, pages 192–201. IEEE, 1999.

[40] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 380–392. IEEE, 2016.

[41] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. Toward standardized near-data processing with unrestricted data placement for gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 24. ACM, 2017.

[42] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2016.

[43] Peter M Kogge. Execube-a new architecture for scaleable mpps. In *1994 International Conference on Parallel Processing Vol. 1*, volume 1, pages 77–84. IEEE, 1994.

[44] Peter M Kogge, Jay B Brockman, Thomas Sterling, and Guang Gao. Processing in memory: Chips to petaflops. In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember at ISCA*, volume 97. Citeseer, 1997.

[45] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 740–753. ACM, 2019.

[46] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *OSDI*, volume 16, pages 705–721, 2016.

[47] John Langford, Alexander Smola, and Martin Zinkevich. Slow learners are fast. *arXiv preprint arXiv:0911.0491*, 2009.

[48] Joo Hwan Lee, Jaewoong Sim, and Hyesoon Kim. Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 241–252, Oct 2015.

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

[50] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2016.

[51] Jiawen Liu, Hengyu Zhao, Matheus A Ogleari, Dong Li, and Jishen Zhao. Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 655–668. IEEE, 2018.

[52] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 367–376. ACM, 2012.

[53] Yuxi Liu, Xia Zhao, Magnus Jahre, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Lieven Eeckhout. Get out of the valley: power-efficient address mapping for gpus. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 166–179. IEEE, 2018.

[54] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. Concurrent data structures for near-memory computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–245. ACM, 2017.

[55] Yun Long, Taesik Na, and Saibal Mukhopadhyay. Reram-based processing-in-memory architecture for recurrent neural network acceleration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(12), 2018.

[56] Patrick J Meaney, Lawrence D Curley, Glenn D Gilda, Mark R Hodges, Daniel J Buerkle, Robert D Siegl, and Roger K Dong. The ibm z13 memory subsystem for big data. *IBM Journal of Research and Development*, 59(4/5):4–1, 2015.

[57] Wei Mi, Xiaobing Feng, Jingling Xue, and Yaocang Jia. Software-hardware cooperative dram bank partitioning for chip multiprocessors. In *Proceedings the IFIP International Conference on Network and Parallel Computing*, 2010.

[58] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, pages 22–31, 2009.

[59] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 457–468. IEEE, 2017.

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

[61] Mark Oskin, Frederic T Chong, and Timothy Sherwood. Active pages: A computation model for intelligent memory. In *Proceedings. 25th Annual International Symposium on Computer Architecture*, pages 192–203, 1998.

[62] Reena Panda, Shuang Song, Joseph Dean, and Lizy K John. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282. IEEE, 2018.

[63] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE micro*, 17(2):34–44, 1997.

[64] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, and Chita R Das. Scheduling techniques for gpu architectures with processing-in-memory capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 31–44. ACM, 2016.

[65] J Thomas Pawlowski. Hybrid memory cube (hmc). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24. IEEE, 2011.

[66] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Reverse engineering intel dram addressing and exploitation. *arXiv preprint arXiv:1511.08756*, 2015.

[67] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security Symposium*, pages 565–581, 2016.

[68] Jayaprakash Pisharath, Ying Liu, Wei-keng Liao, Alok Choudhary, Gokhan Memik, and Janaki Parhi. Nu-minebench 2.0. Technical report, Technical report, Northwestern University, 2005.

[69] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[70] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[71] Vivek Seshadri, Kevin Hsieh, Amirali Boroum, Donghyuk Lee, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Fast bulk bitwise and and or in dram. *IEEE Computer Architecture Letters*, 14(2):127–131, 2015.

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

[73] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 14–26. IEEE Press, 2016.

[74] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 541–552. IEEE, 2017.

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

[76] JEDEC Standard. High bandwidth memory (hbm) dram. *JESD235*, 2013.

[77] Harold S Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, C-19(1):73–78, Jan 1970.

[78] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. The virtual write queue: Coordinating dram and last-level cache policies. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 72–82. ACM, 2010.

[79] Yuliang Sun, Yu Wang, and Huazhong Yang. Energy-efficient sql query exploiting rram-based process-in-memory structure. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2017 IEEE 6th*, pages 1–6. IEEE, 2017.

[80] Zehra Sura, Arpith Jacob, Tong Chen, Bryan Rosenburg, Olivier Sallenave, Carlo Bertolli, Samuel Antao, Jose Brunheroto, Yoonho Park, Kevin O'Brien, et al. Data access optimization in a processing-in-memory system. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, page 6. ACM, 2015.

[81] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 155–166. IEEE, 2014.

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

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

[84] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pages 32–41. IEEE, 2000.