2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00068

Multi-Layer In-Memory Processing

Daichi Fujiki
Keio University
dfujiki@keio.jp

Alireza Khadem
University of Michigan
arkhadem@umich.edu

Scott Mahlke
University of Michigan/Nvidia Research
mahlke@umich.edu

Reetuparna Das
University of Michigan
reetudas@umich.edu

***Abstract*—In-memory computing provides revolutionary changes to computer architecture by fusing memory and computation, allowing data-intensive computations to reduce data communications. Despite promising results of in-memory computing in each layer of the memory hierarchy, an integrated approach to a system with multiple computable memories has not been examined. This paper presents a holistic and application-driven approach to building Multi-Layer In-Memory Processing (MLIMP) systems, enabling applications with variable computation demands to reap the benefits of heterogeneous compute resources in an integrated MLIMP system. By introducing concurrent task scheduling to MLIMP, we achieve improved performance and energy efficiency for graph neural networks and multiprogramming of data parallel applications.**

***Keywords*-in-memory computing, processing in memory, accelerator, GNN**

I. Introduction

Computing systems today have invested the majority of aggregated die area for the memory system. For example, a recent server-class Xeon processor from Intel uses over 70% of its die area for on-chip SRAM caches, and the upcoming Milan-X processor from AMD will have 768MB of LLC [19]. The dense DRAM main memory and NVM-based storage class memory have also played pivotal roles to enable efficient processing of today's data-intensive applications. However, as the amount of data communicated through the memory hierarchy grows, so does the cost of the data movement.

In- and near-memory computing have attracted growing attention for their potential to resolve the disparity between processor and memory performance. Near-memory computing moves processing elements close to the memory, thereby reducing the data movement cost [4], [11], [22], [24], [39], [52], [54], [56], [58], [66], [70]. Moreover, certain memories can morph themselves into compute units by exploiting the physical properties of the memory cells, enabling in-situ computing in the memory array [2], [14], [21], [26], [27], [59], [60], [61]. Compared to near-memory computing, in-memory computing has stricter limitations in data alignment and data movement flexibility, while it can harness the benefits of massive parallelism and reduced data movement, making it suitable for data-intensive/data-parallel applications.

While there is a rich body of work on in-memory computing in each memory substrate, an integrated approach to utilize multiple computable memories in a system has been lacking. As the memory hierarchy today has combined different memories exploiting their trade-offs (e.g., in speed and density), we discover a similar opportunity for in-memory computing.

The wide spectrum of memory technologies and their differentiated compute capabilities open up an interesting opportunity to perform multi-layer in-memory computing in the hierarchy. The preference for in-memory computing is determined by multiple and intertwined factors, such as reuse patterns, data size, and instruction mix [26], [27]. Considering the multiple options of memories, customizing the location of in-memory computing for applications with non-trivial complexity will yield a significant benefit. This is prominent for applications with *runtime workload dynamism*, i.e., the performance determinants (e.g. working dataset size) have a broad distribution and are knowable only at runtime. To maximize the potential of Multi-Layer In-Memory Processing (MLIMP), determining when and where to execute in the memory hierarchy is a challenge.

This work addresses several challenges to enable MLIMP. *First*, considering the state-of-the-art, we design a programming frontend adaptable to multiple in-memory computing frameworks and a memory allocation scheme that allows in-memory computing to co-exist with traditional memory systems. *Second*, we devise kernel mappings of General Matrix Multiplication (GEMM) and Sparse Matrix Multi-Vector Multiplication (SpMM), critical kernels of many ML frameworks, paying attention to maximizing its reuse and resource utilization. As a representative case study, we show the advantages of MLIMP using Graph Neural Networks (GNNs) [12], [33], [41] which entail significant workload dynamism during processing subgraphs. Further, we analyze case studies for multiprogramming scenarios using data parallel applications studied in the prior work. *Third*, we design a scheduler and a performance predictor that are essential to perform efficient job scheduling and fully utilize the resources in MLIMP. Job scheduling in MLIMP is classified into an NP-hard resource constrained project scheduling problem. Also, memory allocation size has to be adjusted to balance parallelism and per-job latency. Based on an analytical scaling model, we develop efficient heuristics to schedule jobs in heterogeneous in-memory systems. Further, to provide an estimation of performance for a specific configuration, we propose a light-weight performance predictor using neural network based regressors. We observe that taking full advantage of multi-layer in-

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00068

memory computing is not possible without introducing sophisticated job scheduling.

In summary, this work offers the following contributions:

- We design MLIMP that re-purposes multiple memories in the memory hierarchy on demand for applications with workload dynamism and diverse compute intensity. The proposed architecture offers a common programming interface and the ability to co-exist in-memory computing with a general cache or memory system.
- Efficient job processing in MLIMP cannot be accomplished without careful job scheduling and performance prediction to maximize the resource utilization. We develop heuristics for the job scheduler using an analytical scaling model and a neural network based performance predictor.
- We show MLIMP can improve general data parallel applications. We also design a kernel mapping of GEMM and SpMM for each memory. We conduct an interesting case study of GNNs demonstrating a significant performance benefit from MLIMP.
- We compare our MLIMP system with a server-class GPU connected to a Xeon processor. Our experimental results show MLIMP can provide an overall speedup of 4.8× for GNNs, achieving 77% of the oracle throughput. General applications also achieve 7.1× speedup compared to single layer IMP. The proposed architecture improves the energy efficiency by 5.02×.

II. Motivation and Background

A. Motivation

There is a significant body of research on in-memory computing with individual layers of memory. Figure 1 shows the relative energy per access, delay, and metrics for calculating available compute parallelism of different memory technologies. The parallelism can be estimated based on available sense amplifiers (SAs) at the bitline peripherals per unit area. This is because each bitline operation usually requires sensing at SA to complete. It is also dependent on the bit-cell structure and design target (e.g., cache, main memory, or storage DIMMs). For example, while NAND-Flash and DRAM have a small cell size, their available parallelism can be low because a large number of cells in an array share the same set of sensing amplifiers (low SA density).

Computing with Non-Volatile Memories (NVMs) has different trade-offs compared to computing with SRAM or DRAM. Since NVMs are more stable against data corruption, they can support operations involving multiple wordlines. Due to their high density, NVMs can accommodate large datasets which dwarf SRAMs. Higher density also increases data level parallelism of in-place computation. On the other hand, in-memory computing in NVMs (STT-RAM and ReRAM) can be one to two orders of magnitude slower and requires significantly higher energy per bit when compared

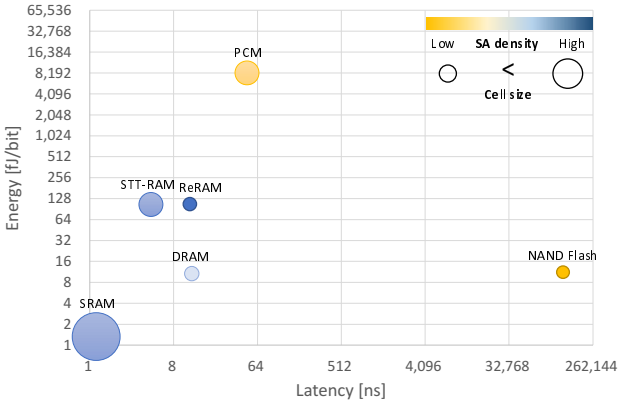


Figure 1. Energy, latency, and parallelism characteristics of various memory technologies.

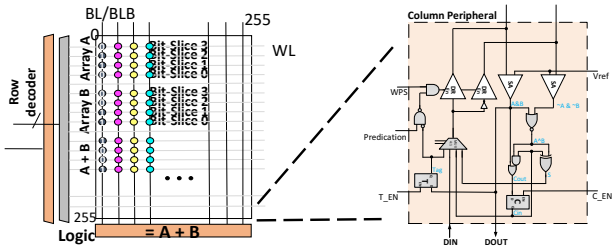
to SRAMs. Further, NVMs have limited endurance (and high write energy/delay) which curtails the number of writes the memories can reliably sustain. Similarly, DRAMs pose their own unique challenges, such as destructive read access and stability against data corruption.

While prior research has proposed a variety of in-memory computing approaches in individual memory, a framework that integrates multiple computable memories into the memory hierarchy has been lacking. Given the wide spectrum of memory technologies and their differentiated compute capabilities, customizing the memory hierarchy for specific application domains may yield significant benefits.

B. In-Memory Computing

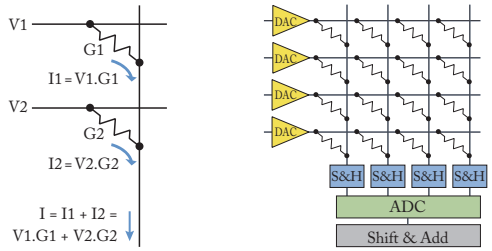
1) In-SRAM Computing: In-SRAM computing activates multiple wordlines of SRAM arrays and performs logic or arithmetic operations on vertically aligned bit cells within a column. Compute Caches [2] introduces an in-SRAM computing framework that supports copying, zeroing, XOR, comparison, and search. Multi-row activation produces NOR and AND of two bit cells at the end of the bitline (BL) and bitline bar (BLB). BL and BLB are sensed independently by a re-configurable differential senseamp [2]. Combining the results with extra logic gates at the peripheral, any binary commutative operations, including the universal operator NAND, can be derived.

Logic operations can be sequenced to perform arithmetic operations. Neural Cache [21] supports arithmetic operations inside the SRAM arrays for machine learning workloads, and Duality Cache [27] further extends it for floating point operations for general data parallel applications. They vertically align operands in each bitline and perform computation in a bit-serial manner. As opposed to bit-parallel computing which processes multiple bits in a single data word, bit-serial computing processes bit-by-bit, taking multiple cycles to produce results. Each n-bit element is stored across n wordlines, and thus each wordline holds one *bit-slice* of 256 vector elements as shown in Figure 2 (a).



(a) 8kB SRAM array (b) Bitline ALU

Figure 2. In-SRAM computing (Neural Cache [21]).



(a) Multiply-accumulate operation (b) Vector-matrix multiplier

Figure 3. In-ReRAM computing (adopted from [60]).

A 1-bit *full adder* can be implemented using a few gates at the peripheral as shown in Figure 2 (b). By adding each bit iteratively, we can perform the addition of two n bit numbers in n cycles. Multiplication takes n^2+3n-2 cycles and is implemented as a series of additions of partial products. Due to the dense SA density of SRAM, LLC of typical server class CPU can be transformed into millions of bit-serial ALUs.

2) In-DRAM Computing: Prior work has mainly focused on near-memory computing using DRAM, including 3D stacked memory and bit-serial ALUs attached to each bitline or sense amplifier ([23], [46]) There has been several known obstacles for DRAM-based *in-memory* computing, such as logic cost and memory density issue. Charge sharing techniques are proposed as a key enabler of DRAM-based in-memory computing [28], [47], [59], [65]. Charge sharing techniques activate more than one wordline and perform bitwise operations by exploiting altered charges in capacitors connected to the same bitline. Hence, it can provide some important logic operations with a small area cost.

Ambit [59] proposes charge sharing based bitwise AND and OR operation. Ambit simultaneously activates three wordlines (referred to as triple-row activation or TRA), and based on the charge sharing principles [38], the status of vertically aligned three cells is determined. The behavior of TRA is the same as a 3-input majority gate. By using one cell as a control bit C, TRA can perform AND ($C=0$) and OR ($C=1$). Ambit also supports NOT operation using a dual-contact cell that has an additional transistor. A combination of AND and NOT forms NAND, a functionally complete operator. Therefore, Ambit can support any logical operations and arithmetics [59].

3) In-ReRAM Computing: The linear IV characteristics of ReRAM cells are exploited for in-memory computation in the analog domain. In-ReRAM computing feeds reference voltage under the threshold for set and reset, and the bitline current that results from this operation is interpreted as the outcome of the multiplication of cell conductance and the input voltage. Furthermore, by activating multiple rows, currents that flow from different memristor cells sharing a bitline accumulate in the bitline, following Kirchhoff's law, as shown in Figure 3 (a). This analog computing capability of memristors is leveraged for accelerating machine learning workloads of which computation is dominated by multiply-accumulate (MAC) operations that compose dense matrix multiplications [14], [60], [61] (Figure 3 (b)).

IMP [26] proposes a programmable in-memory processor architecture and data-parallel programming framework. Data parallel applications are described using TensorFlow and compiled for the ReRAM crossbar array. IMP supports various integer operations leveraging extra circuitry and the compiler's lowering operations.

C. Acceleration targets

1) Data-Parallel Applications: Prior work on in-memory computing has mainly focused on accelerating machine learning workloads such as DNNs and CNNs by efficiently performing GEMM and dot-product in situ in memory. On the other hand, the compute capability of memories is also explored for data-parallel applications. IMP [26] runs SIMD vectorized kernels from Parsec and Rodinia benchmarks, and Duality Cache [27] supports SIMT applications written in CUDA and OpenACC.

The execution time of the compute kernels in data-parallel applications depends on the in-situ operation latency and parallelism of each memory and the instruction mix of the applications. Therefore, they can have different preferences for memory. Moreover, given multiple such data-parallel applications running concurrently, one has to carefully choose where to execute each job, so that one memory will not be oversubscribed and the turnaround time will be minimized. As these applications have a variety of execution times, it is important to think about a sophisticated job dispatching and resource allocation scheme.

2) GNN (GEMM + SpMM): GNNs [12], [33] revolutionalize commertially and academically important inference tasks based on a graph structured data, such as recommender system [1], [10], protein interaction prediction [57], and drug response prediction [63]. GNN is an algorithm applied to a graph G.Each node v has its input feature vector x_v. At each layer, the node features are propagated to its neighbors and get updated. Thus, output node features (also referred to as node embeddings) of the k-th layer include information from k-hop away neighbors. The most important kernels of GNNs

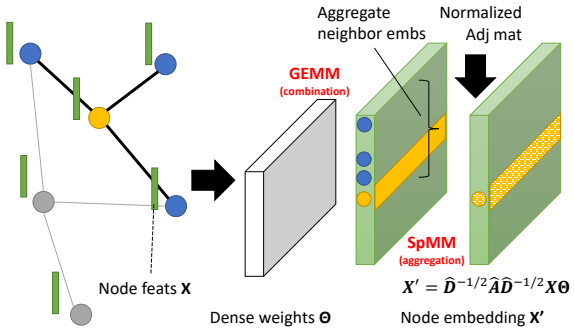


Figure 4. Operations in GCN

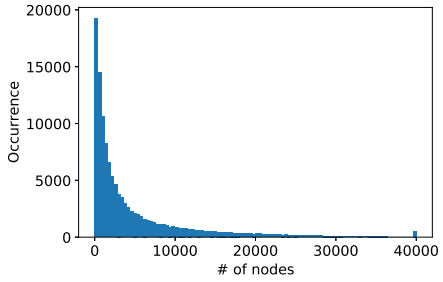


Figure 5. Node distribution of 3-hop subgraphs in ogbl-citation2 dataset.

lie in this *iterative update* step, and it is manifested by an *aggregation-combination* function, as shown in Figure 4.

During aggregation, the feature vectors from neighboring nodes and the feature of the node/edge itself are *aggregated* by functions such as mean and max to a single feature vector. Using a matrix representation, it does $B = \overline{A}X$, where $\overline{A}$ is the normalized adjacency matrix ($= \hat{D}^{-1/2}A\hat{D}^{-1/2}$, D is the diagonal degree matrix [41]) and X is the feature matrix. Since $\overline{A}$ is a sparse matrix and X is a dense matrix, aggregation performs SpMM, which is known to have an irregular memory access pattern. Thus the key kernel for aggregation involves SpMM. The aggregated feature vector will be *combined* to yield a new feature vector. The new feature vector is used as an input for the next step. The combination function is typically composed of a feed-forward network. Hence, the main kernel of the combination step uses GEMM.

Many state-of-the-art GNN frameworks employ an approach called subgraph learning (a.k.a. mini-batching) [68], [69] to efficiently train on a large-scale graph [15] and to improve accuracy by inducing regularization effects [35]. It extracts the k-hop neighborhood of nodes of interest and applies GNNs to this subgraph.

While subgraph learning shows promising results, it induces substantial variation in the working dataset size and compute load. The subgraph size distribution of a real-world graph is shown in Figure 5. Since the processing time of in-memory computing is correlated with the subgraph size, the variation is also propagated to the latency. Thus, a monolithic approach using a single type of hardware or memory-centric acceleration (e.g., in-SRAM only) will be suboptimal. In this context, we observe that a sophisticated job scheduling is crucial to harnessing the heterogeneous computation resources of MLIMP.

III. MLIMP

In this section, we present the MLIMP system stack. The architecture of MLIMP is presented, then we introduce our job scheduling approaches to enable optimized job processing. Following that, we show our kernel mappings of data-parallel applications and GNNs, and propose a performance predictor for efficient scheduling.

A. Overview

The proposed system is shown in Figure 6. At **runtime**, a call to a function that has been explicitly marked for in-memory processing triggers the MLIMP scheduler. Such a function call generates MLIMP jobs. The scheduler creates an optimized schedule with inputs from the performance predictor, and enqueues jobs into individual queues for each memory layer. Note that the scheduler and job queues are implemented in the system software. The functions used for in-memory processing are data-parallel kernels with no side-effects (i.e. no implicit sharing of variables with other parts of the application). There is no context or task migration from the host processor. Our current execution model accesses input data via SIMD load instructions. Future work can possibly extend the execution model to support other forms of function call ABI, e.g. x86-64 variadic function call ABI. The in-memory device has support for storing cross-compiled binary code similar to prior in-memory architectures and the execution is launched by the system firmware. The runtime flow is similar to the kernel launch for CUDA runtime. Indeed, in-memory processing can be viewed as a tightly coupled data-parallel accelerator with a runtime and an execution model similar to GPUs.

At **compile time**, the in-memory processing function targets are described in SIMD data flow graph (DFG) [26] and cross compiled for different target in-memory ISAs, as shown in Figure 6. SIMD DFG can be programmed in general programming languages and extracted from intermediate representation or directly from tensor computation frameworks (e.g. TensorFlow can dump DFG in the protobuf format). The gaps in the supported operations between the frontend and ISA are bridged by the compiler's lowering and legalization operations. This enables operation level abstractions such as matrix operations to work with existing machine learning frameworks, and also facilitates expressing data-parallel kernels with a common programming front-end. At the execution time, a suitable in-memory device is chosen by the scheduler and resources are allocated. In this way, architecture-specific optimizations (e.g., VLIW execution of [27]) and algorithm-level optimizations (e.g., kernel

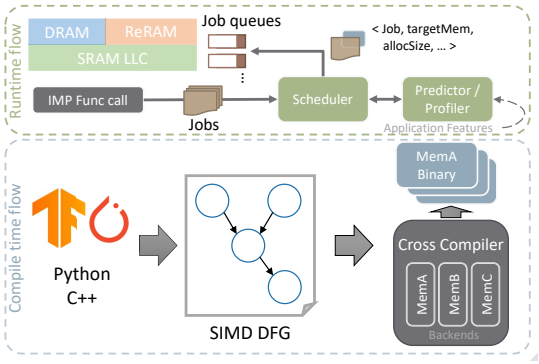


Figure 6. MLIMP framework. SIMD DFG [26] of the target kernel is extracted and cross compiled by backend compilers which target different in-memory layers and ISAs. At the runtime, a function call generates MLIMP jobs which are scheduled for execution on different in-memory layers with favorable memory allocations.

execution order) can co-exist. Note that other programming models (e.g. CUDA [27]) are possible. This work assumes SIMD data flow graph as a versatile programming model for both graph neural networks and general data-parallel applications.

B. Architecture Support

1) Common Programming Interface: Prior work has covered most of the innovations needed to enable in-memory computing in the existing memory hierarchy as described in Section II-B. To interface with the heterogeneous in-memory computing resource efficiently, we need to design a common programming frontend.

While the instruction set architecture (ISA) and the preferred data mapping within an array vary for each memory (e.g., in-SRAM computing performs bit-serial computing on vertically aligned data, while in-ReRAM computing uses bit-parallel computing with multi-level cells), most of the in-memory computing works support arithmetic operation level abstraction either in their API or ISA. They usually support integer arithmetic operations, and some also support floating points. For wide compatibility with the past proposals, we focus on integer operations in this work.

Binary bit-serial computing with bit transposed data is employed for in-SRAM and in-DRAM computing. To make peripheral complexity comparable, memory arrays compute a universal operator such as NAND and NOR with the smallest possible cycle counts, and the result and any byproducts (e.g., AND) are fed to extra logic gates at the peripheral to perform the rest of the operations. In-ReRAM computing performs bit-parallel computing with the peripheral shifter and adder, and extra logic such as LUTs is introduced to enable other non-native operations.

Taking the intersection of supported arithmetic operations among the three types of in-memory computing devices, our programming interface supports integer addition, subtraction, multiplication, division, comparison, moves, and simple transcendental functions (e.g. exp2). The arithmetic operations abstracted in each ISA are further expanded into a sequence of micro-operations within controllers or FSMs in each memory [21], [27], [59].

2) Memory allocation: Memory workspace for in-memory computing is allocated within a scratchpad memory region in each memory to ease the collocation with the existing memory virtualization frameworks. There is a body of work trying to enable private scratchpad memory within the cache and main memory [18], [44]. This is a middle ground approach of two extremes: using all memory space as a scratchpad for in-memory computing (most of the prior work) and completely integrating in-memory computing with the existing memory management system.

Complete integration would enable seamless processing of in-memory operations with minimized data copy and transformation. While there are non-trivial benefits to the complete integration of in-memory computing under the existing memory management system, its cost is also non-trivial. Supporting compute cache lines and other cache lines in a finer-grained manner under the general set-associative cache scheme would lead to a prohibitive cost for guaranteeing data layout and bookkeeping the cache lines for avoiding unexpected cache line replacement, etc. In contrast, the hybrid approach using VLS [18] enables scratchpad memory on a *coarse* partition of cache (e.g., a single way) with a tiny modification to the cache architecture.

It is possible in the main memory to align data to an exact position of a physical memory array by reverse-engineering the XOR-based address mapping of microarchitectures [55] and by modifying operating systems' memory management system to support finer-grained page coloring [16]. However, it comes at the cost of expensive page management (i.e., page numbers for each color have to be extensively searched [16], [17]) and involves a risk of external fragmentation. We thus consider the complete integration of in-memory computing with the current memory virtualization scheme does not provide convincing benefits compared to its cost, but future work can address this issue. The hybrid approach still allows compute regions to co-exist with existing managed memory space in a coarse grained manner, while guaranteeing data layout flexibility that is essential to in-memory computing.

C. Scheduler

1) Scheduling Challenges: Conventional computers perform job scheduling in their operating system (OSs). We notice that the existing job scheduling approaches are not directly applicable to in-memory computing. While OS job scheduling targets a homogeneous CPU architecture, we need to choose from a variety of in-memory processors. Although there exist OS job schedulers for heterogeneous cores (e.g., big.LITTLE architecture [7]), in-memory computing is additionally required to determine the allocation size of

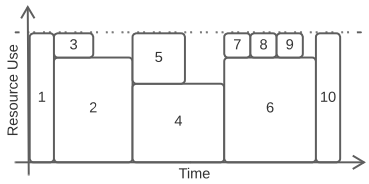


Figure 7. Resource Constrained Project Scheduling Problem (RCPSP). Multi-layer in-memory computing has another dimension for memory type.

the memory. Execution time is also largely dependent on memory properties and jobs; thus, simple scaling techniques (e.g., big cores are about x times faster than small cores) cannot be applied. These three factors, i.e., memory type, allocation size, and challenges in estimating execution time, make it difficult to apply OS's job scheduler for in-memory computing.

In fact, the job scheduling problem of this set-up is categorized into NP-hard Resource Constrained Project Scheduling Problem (RCPSP) [43]. As illustrated in Figure 7, Scheduler has to choose the right resource amount (and resource type) for a job, as well as its execution order. Multiple jobs can be executed at a time upon the availability of resources. While there is a rich body of work from the operations study community [42], [43], [53], there is no known golden solution to RCPSP (except for a special case of Johnson's rule [36]), so the problem needs to be approached on a case-by-case basis [51].

2) Baseline - Longest Job First Scheduling: For our baseline, we use the Longest Job First (LJF) scheduling. LJF scheduling tends to increase work in progress while making short jobs late [6]. This is ideal for minimizing the batch process time. Short jobs being late does not matter because all jobs in the batch need to wait for the completion of batch processing before moving to the next step. Rather, increasing in-flight jobs is important because it improves resource utilization. Moreover, if jobs in one processor end earlier, LJF makes it easier to load balance by filling the gap with smaller jobs left in the queue. Our schedules are thus based on LJF.

Each job in an incoming batch is first processed by a performance predictor (covered in Section III-E) to calculate the estimated execution time for each in-memory processor. The baseline LJF scheduling does *not* adjust the memory allocation size, but uses a fixed size $a_{unit} = max\_size/P$ where P is the number of parallel jobs. Then, jobs are pushed into a single queue in descending order of the shortest execution time. Whenever a spot is available, a job item at the head of the queue is dequeued and scheduled to the best performing memory.

3) Scheduling with Variable Memory Allocation: The memory allocation size can be adjusted to optimize in-memory compute performance. While applications with a large dataset may benefit from a large allocation, allocating as large memory as the entire working dataset can be suboptimal when the compute intensity for the allocation is low (e.g., SpMM of a very sparse matrix). On the other hand, a small memory footprint but compute-heavy workload can benefit from a larger allocation by performing data replication and parallel operations on data replicas. Replication copies data within memory, reducing off-chip bandwidth of applications with a data reuse opportunity [21].

To seize this opportunity, we need to know the memory allocation size for a job that minimizes execution time. This requires an understanding of the relationship of the allocation size with the execution time. We develop a analytical performance model which determines the execution time $t(x,m)$ for job x with allocation size m. The proposed scheduler calculates the tradeoff of execution time and memory allocation size using an analytical performance model, and then determines the best memory allocation size for a specific job.

Our performance model is composed of two parts, load time $t_{ld}(x,m)$ and compute time $t_{cmpt}(x,m)$, and the expected job execution time t for job x with allocation size m is calculated as the product of the number of iterations and the sum of the latency from the two parts:

$$t(x,m) = n_{iter}(x) \times (t_{ld}(x,m) + t_{cmpt}(x,m)). \quad (1)$$

Let $a_{repunit}$ be a unit allocation for one replica and the static dataset size of the kernel. If the whole working set does not fit in the allocated memory, the number of iterations $n_{iter}(x) = datasize(x)/a_{repunit}$ becomes larger than 1. The load latency t_{ld} is calculated based on the time to load input data and the time to replicate data. The number of replications is calculated by $m/a_{repunit}$. Thus, we have

$$t_{ld}(x,m) = t_{ld}(x) + t_{replica}(m/a_{repunit}). \quad (2)$$

The compute performance model that estimates $t_{cmpt}(x,m)$ assumes the *scale free* property [8] of resource size and performance. In our case, the resource size is memory allocation size m. With a shape parameter β, our scale-free model is described as

$$t_{cmpt}(x,m) = t_{cmpt}(x,a_{repunit}) \left(\frac{a_{repunit}}{m}\right)^{\beta}. \quad (3)$$

The intuition is that t_{cmpt} is inversely proportional to the amount of parallel processing possible due to replication and the number of in-memory processing elements available. The parallelization cost can be empirically modeled by setting the shape parameter β less than 1. The performance for the unit allocation $t_{cmpt}(x,a_{repunit})$ is provided by the performance predictor covered in Section III-E or through profiling. Similar to load latency, if the whole working set does not fit in the allocated memory, the compute latency is multiplied by the number of kernel iterations $n_{iter}(x)$ as in Equation 1.

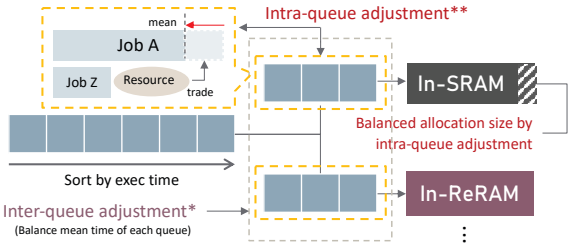


Figure 8. Inter-Queue* and Intra-Queue** Adjustments.

We observed that the scale-free compute model fits well to a variety of problems. For example, SpMM kernels in the Open Graph Benchmarks (OGB) [35] sees a median R^2 of 0.998. There is a small deviation in the compute performance model for the combination of small-sized jobs and large memory allocation due to a lack of enough parallelism to exploit all allocated resources. Fortunately, this minimally affects the overall performance because such small-sized jobs take a short time to finish execution and it is unlikely for small jobs to get large memory allocation from the scheduler.

The scheduler finds the memory allocation size for a job by finding the best m to minimize $t(x,m)$. We find that the memory allocation size m that perfectly minimizes $t(x,m)$ tends to overprovision resources as the execution time curve flattens out with large memory allocation sizes. To solve this problem we use m that roughly corresponds to the knee of the execution time curve $t(x,m)$ plotted w.r.t memory allocation size. Precisely, the scheduler uses m that maximizes the angle speed $\partial\theta/\partial m$ of the tangent to the execution time curve.

4) Adaptive Scheduling: To balance the execution time of the multi-queue LJF scheduling, we introduce *inter-queue adjustment* shown by Algorithm 1. The goal of the inter-queue adjustment is to balance the mean execution time between queues (Figure 8 middle). For each iteration, it calculates the mean processing time of the job items in each queue. If the maximum difference of the mean times is larger than the acceptable gap ε, it migrates *migr_cand*, the job with the smallest execution time (when executed in *min_mem*), from the *max_mem* queue to *min_mem* queue. This is repeated until the mean time difference is below ε or migration no longer contributes to improvement in job balancing. After successful inter-queue adjustment, proper resource distribution will lead to an execution time close to the mean.

Adaptive scheduling dispatches jobs in the queue in a greedy fashion. Whenever there are available resources that can run a job with its requested allocation, it runs the job, giving priority to larger jobs. If there are any remainder resources not allocated by the prior procedure, the scheduler calculates the expected completion time for each awaiting job in the queue and dispatches jobs if they can finish earlier than the completion of jobs in flight using the remainder resources.

Algorithm 1 Inter-Queue Adjustment.

Input: *queues*: Map[mem, queue], t_{mem}: $x \rightarrow t_{mem\_unit}(x)$

for up to N times **do**
  $\bar{t}$ = {*mem* : get_mean(*queues*[*mem*]) foreach *mem*}
  Get *max_mem*, *max_mean*
  Get *min_mem*, *min_mean*
  if difference(*max_mean*, *min_mean*) $> \varepsilon$ **then**
    migr_cand = $\arg\min_{x \in queues[max\_mem]} t_{min\_mem}(x)$
    Migrate *migr_cand* from *queues*[*max_mem*] to *queues*[*min_mem*] if $\bar{t}$ improves else **break**
  else
    break
  end if
end for
return *queues*

5) Global Scheduling: Adaptive scheduling can flexibly adjust the dispatching order even if there is a gap between the estimated execution time and the actual time. However, it is challenging to fully utilize the resources due to scheduling bubbles. Bubbles are introduced when a small remainder allocation cannot be utilized by any waiting jobs.

The global scheduler adjusts the allocation size in each queue to fully utilize the resources and generates a complete job dispatching schedule in advance. Instead of directly using the provided resource allocation, the global scheduler further adjusts the allocation size using the *intra-queue adjustment* algorithm in Algorithm 2. The objective of the intra-queue adjustment is to balance the time of long jobs, which can take longer than the mean execution time, by trading the resources from the smaller jobs in the queue as shown in Figure 8.

For each queue, the intra-queue adjustment finds the largest and smallest job, and if the largest job takes more time than the mean, it calculates the allocation size necessary to achieve the mean execution time. The difference is migrated from the smallest job's allocation, as long as minimum resources are left. It repeats this process until all jobs can finish within the mean execution time. In a rare case with a large discrepancy in the job size distribution, the longest job cannot achieve the mean even when setting the minimum allocation to the other jobs.

We observe that global scheduling can achieve better performance under the circumstances where the predicted execution time is precise because of better resource utilization and fewer bubbles. Thus, the choice of the adaptive or global scheduler will be determined by the accuracy of the performance estimation.

D. Kernel Mapping

1) Data-Parallel Applications: While there are several execution models for general data-parallel applications for in-memory computing, we use wide SIMD applications

Algorithm 2 Intra-Queue Adjustment.

for each *queues* **do**
  for up to N times **do**
    Sort *queue* based on $t(x, z(x))$
    Get $max\_x, max\_t, min\_x, mean\_t$
    if difference(max_t, $mean\_t$) $> \varepsilon$ **then**
      $swap\_cnt = t^{-1}_{max\_x}(mean\_t) - m_{max\_x}$
      $swap\_cnt = \min(swap\_cnt, m_{min\_x} - m_{minimum})$
      break if $swap\_cnt == 0$
      Migrate $swap\_cnt$ of resources from min_x to max_x
    else
      break
    end if
  end for
end for
return *queues*

in IMP [26] for simplicity. The kernel mapping of IMP applications follows their vectorized VLIW execution model, which can also be adopted for in-SRAM computing [27]. We extract their compute kernels and compile them for different in-memory processors, performing static analysis to obtain the execution time for each code block. The statistics are used by the scheduler. At the execution time, it chooses the best in-memory processor based on the scheduler output.

2) GEMM: General Matrix Multiplication (GEMM) is a core kernel of many machine learning frameworks. Prior work has proposed efficient GEMM operations in memory [14], [21], [60], [61]. For example, in-ReRAM computing can perform vector-matrix multiplication using analog multi-operand MAC computation. The weights are stored in memory and reused across different inputs. The compute-efficient data mapping of weights varies according to the memory. For example, in-ReRAM computing generally employs a natural 2D mapping of the weight matrix. Each value can use multiple memory cells to improve precision [60].

Bit-serial computing does not generally support multi-operand operations. Thus, it is crucial to exploit parallelism in the architecture efficiently. For example, Neural Cache [21] unrolls input activation of CNN for each sliding window and duplicates it for each output channel. We take a similar approach for GEMM. The weight matrix is serialized to a vector representation and stored in the topmost register of each SIMD slot. The input feature vector is duplicated for each column of the weight matrix and stored in a SIMD slot with a corresponding weight multiplicand. In this way, all multiplication operations can be done in parallel for each input feature vector. Then, reduction operations are performed to make sums to complete dot-product operations. A memory array can have multiple input feature vectors. The weights can also be replicated to fully utilize the available memory space.

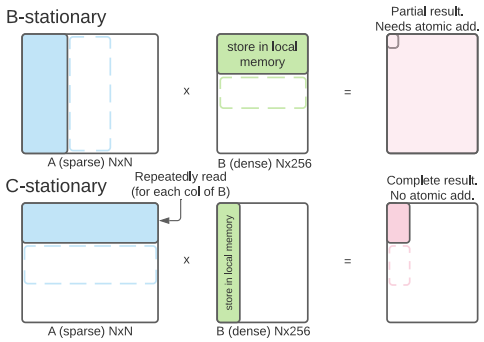


Figure 9. Data reuse patterns of SpMM.

3) SpMM: **Sources of Inefficiency:** Sparse Matrix Matrix Multiplication (SpMM) multiplies sparse matrix A and dense matrix B. In-memory computing in general is less efficient for sparse computation due to the random scattered access patterns of the workload. While in-memory computing generally requires operands to be arranged in a designated location to perform computation, this scheme cannot be directly applied to a compressed storage format of a sparse matrix. To expose the computation models of in-memory computing, sparse matrices need to be decompressed to the dense format, reinserting null elements eliminated by the compression. Existing work on in-memory graph processing accelerator [62] also performs decompression of the CSR format to perform Sparse Matrix-Vector multiplication (SpMV), which many graph algorithms can be transformed into.

Decompression leads to the following inefficiencies: (1) inevitable data movement and its related cost and write count, (2) low compute density per array which undermines the throughput oriented in-memory compute resources, and (3) low locality due to irregular vertex access patterns, which may result in repetitive decompression due to capacity limitation of memory.

Lookup-Based Approach: For these reasons, we store the dense matrix B in the memory array to bypass the computation density issues related to the sparse format. B is partitioned into horizontal slices and stored into arrays. Then, a corresponding vertical slice of the sparse matrix A is loaded from the main memory and processed row-by-row. If A is a binary adjacency matrix, arrays will perform a series of vector additions of some rows of B, using non-zero column indices of A as indices to look up the B rows. If A has non-binary values (e.g. edge weights), arrays will instead perform a dot-product computation using the A value as the multiplicand. Buffer arrays are utilized to temporarily store and accumulate the partial sum vector from multiple arrays, playing a similar role as a reduction tree.

Reuse Model: As illustrated in Figure 9, there are several reuse patterns for SpMM [25]. While CPU and GPU generally employ the C-stationary approach [25], [34], we adopt B-

stationary. B-stationary maximizes the reuse of the dense B matrix, which is ideal for subgraph learning of GNNs because it is known that all the node features are reused several times while processing the batch. Exploiting the predetermined knowledge of feature reuse, in-memory computing can fully utilize the locality of node feature access. On the other hand, B-stationary requires an atomic update of the results if multiple processors can add partial sums to the same output elements. However, given the skewed non-zero distribution of real word graphs, not every slice pair of A and B contribute to one output position, thus the cost of update is not significant, and is better than multi-loading A (Figure 9, C-stationary).

B-stationary also provides an opportunity for efficient vector processing. In contrast, C-stationary needs to perform lengthy reduction operations with a lot of null entries to make a complete output. We observe B-stationary achieves 4.3x better memory latency performance and 42x better compute performance (obgl-collab [35]).

Replication: We also replicate the B slices within the memory allocation, reducing the slice size accordingly. This is to leverage the input row parallelism by performing parallel reductions or dot-product operations for different rows of A. Since the input row parallelism is easy to find, we find that having a few replicas helps achieve good performance scaling.

E. Performance Prediction

The performance predictor predicts the expected execution time of a job and is an essential component of our scheduler. Compute time ($t_{cmpt}(x,m)$) for a basic block of most of the in-memory workload studied before can be deterministically calculated at the compile time. In such a case, while input parameters can affect the number of kernel invocations, it is straightforward to estimate the performance of each invoked kernel [26]. This applies to GEMM and many data-parallel applications, including those that we evaluate.

On the other hand, the execution time of SpMM is dependent on the contents of the adjacency matrix of the subgraph. This is because the in-memory device also serves as a *memory* for storing features, and its access patterns are dependent on the input adjacency matrix. Here, each access is followed by a vector MAC operation. While the cycles for MAC operations are deterministic, we do not know how many MACs will be triggered. It is possible to know only by a complete scan of the input, which is impossible at the compile time, and even at the execution time, performing a full scan of the adjacency matrix for cycle estimation becomes costly.

Limitations on a Simple Metric: Job size per allocation unit can be used as a proxy to estimate the execution time for such workloads. Let us first verify this claim. For SpMM, the job size within a given allocation can be calculated from the number of non-zero partial rows (prows) of width w in the adjacency matrix. Prows are rows in vertical strips (Figure 9 top) and non-zero prows are such rows with at least one non-zero element. Let $H_w(x)$ be a function that returns the number of non-zero prows of a subgraph x of width w, then the average amount of job per allocation (translated into w) can be calculated from $nnz(x)/H_w(x)$.

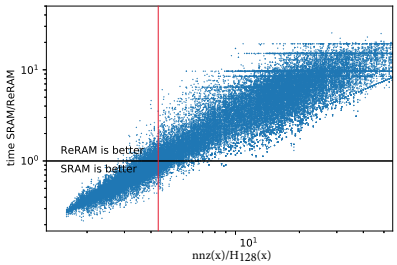


Figure 10. A naive classification model using $nnz(x)/H_{128}(x)$ as a metric. Red line is a threshold.

Figure 10 shows the memory preference t_{SRAM}/t_{ReRAM} for different jobs plotted with $nnz(x)/H_w(x)$ that is used as the metric. We can see that ReRAM outperforms when the job size per allocation is large, i.e., the access is likely to be localized and there are lots of opportunities to perform the multi-operand operations in an array. This trend is reasonable because ReRAM has a larger register capacity per array and can perform a multi-operand dot product operation. Although $nnz(x)/H_w(x)$ is correlated to the memory preference and thus can be used to roughly classify jobs, there are a lot of borderline jobs that are misclassified. Also, a complete scan is necessary to know $H_w(x)$.

Our Approach: We thereby use MLP regressors to give a better classification for this non-linear classification task, and also to generate an estimated time for each memory. A similar approach is adopted in a prior work that used an MLP regressor and classifier to make the best selection of matrix permutation for SpMM [49]. Job size and performance can be correlated with a set of subgraph metadata, given the subgraphs are generated from the same *mother* graph, based on the *scale-free* property of the real-world graphs. We use two MLP regressors to learn H_w and cycle counts from the graph metadata. For each mother graph, H_w is first trained, taking w, the dimension of a submatrix, and nnz as the input from the training subgraphs. Then we use predicted $H_w(x)$ and the same set of metadata to train another regressor for the cycle counts. The regressors have two hidden layers with 16 and 8 nodes. While MLP regressors are simple, the cycle count predictor can achieve relatively good accuracy (e.g., R^2 score of 0.995 and RMSE of 22% of the mean cycles for ogbl-citation2 in SRAM). Note that the training cost is one time for the mother graph, and the regressor model can be reused for all input queries.

We notice that using more hidden nodes/layers in MLP does not significantly contribute to the accuracy. Random forest based solutions such as XGBoost [13] regressor can

Table I
DATASET DETAILS.

Dataset	#Vertex	Input/hidden feature	#Edges	Raw datasize	Min. req. memory
ogbl-collab	235,868	128/256	1,285,465	293M	5GB
ogbl-citation	2,927,963	128/256	30,561,187	3.8G	40GB
ogbl-ppa	576,289	58/256	30,326,273	340M	2GB
ogbl-ddi	4,267	- /256	1,334,889	9.5M	2GB
ogbn-products	2,449,029	100/256	61,859,140	3.4G	33GB

Table II
DATA PARALLEL APPLICATIONS.

		App Combinations						
Application	Domain	A	B	C	D	E	F	G
Blackscholes	finance	✓		✓		✓		
Fluidanimate	fluid dynamics	✓		✓				✓
Streamcluster[A,B*]	data mining	✓[A]	✓[B]		✓[B]	✓[A]	✓[B]	
Backprop	pattern recog		✓		✓			✓
Kmeans	data mining		✓				✓	✓
Crypto	message auth	✓			✓		✓	
DB[B,S**]	database			✓	✓[B]	✓[S]	✓[B]	
Bitap	string search		✓			✓		✓

*Streamcluster has two input data size, A and B.
*DB has two algorithms: bitmap index (B) and full scan (S).

achieve up to 2x better accuracy (RMSE), while requiring significantly more computation and parameter storage cost compared to MLP.

F. Generality of Our Approach

Our scheduling approach can be broadly applied to various data-parallel applications. Similar to OpenMP or CUDA programs, there are largely two approaches to generating parallel jobs: (a) generating a fixed number (e.g. core count) of jobs with a dynamic loop count and (b) generating an input-dependent number of jobs with a fixed loop count (fixed load per core). MLIMP supports both approaches. Our SpMM takes the former approach, using a predictor to predict latency for each dynamic job. While predictor needs pre-training, it performs best for applications with fused memory access and compute operation. On the other hand, general data parallel applications can take the latter approach, where the job performance can be estimated simply by profiling. Instead of training a predictor, the scheduler can use profiling results under unit resource allocation and scale them. In both cases, our scheduling approach will try its best to increase the throughput.

IV. METHODOLOGY

Benchmarks: We show the compute kernels for tested data-parallel applications in Table II. In addition to applications from IMP [26], we use applications with bulk bitwise operations. Crypto is a kernel in SipHash [9], a cryptographic hash used for message authentication code. DB is database search queries using a bitmap index and full scan. Bitap is a string search algorithm widely studied in bioinformatics workloads etc. Each application generates multiple jobs with a fixed loop count. We compare the kernel execution time of each application. The kernels are compiled for target machine configurations of each memory, adopting the SIMD VLIW execution model of the prior work [26], [27]. The machine configuration of ReRAM is taken from IMP [26] and that of SRAM is taken from Duality Cache [27]. For both targets, the latency of the compute kernels can be calculated deterministically and we use this profile as the input to the scheduler.

To evaluate machine learning kernels, we use a GNN framework with three Graph Convolutional Network (GCN) [41] layers. The models and graphs are from Open Graph Benchmarks (OGB) [35] (see Table I). GNN input features and weights are trained for 16bit fixed-point precision with an additional feed-forward network. This quantization only results in a slight accuracy degradation of $< 1\%$. All GNN workloads are built on top of PyTorch framework and PyTorch Geometric (PyG) libraries that are compiled for both CPU and GPU. Subgraphs are generated by PyG's neighbor sampler. We use the autograd profiler of PyTorch and NVIDIA's NVVP and PyProf profilers to generate the execution trace and profiling results on the native machines. We use a batch size of 64. Due to the limitation on the simulation time, we sampled a random 10 batches (640 queries in total) for the simulation.

GCN also contains other operations such as activation functions (e.g., ReLU), but they take insignificant time and are thus executed in the host processor. The building block kernels of our GNN approach are not dependent on each other, so they can be reordered and applied to other GNN frameworks.

Subgraphs in a batch can be precomputed or dynamically generated using a data generator process or a remote graph server. We assume the subgraph data is precomputed [69], but since the workload is similar to breadth first search (BFS), it can be efficiently executed in many near-memory computing enabled memory systems [3], [50], [67].

By default, a batch contains subgraphs for each query input. However, it is sometimes useful to generate a concatenated subgraph that has a union of all nodes in the subgraphs, while giving up the opportunity of job mapping of subgraph granularity. This is when the graph has a high degree of connectivity and the intersection of nodes between k-hop subgraphs is large, which leads to a good chance of reusing node features across different query inputs. We observe the large connectivity in some of the graphs in the nature domain in our benchmark (ogbl-ppa and ogbl-ddi), so we take this approach.

Performance and Power Models: We develop an event-driven simulator with timing models from IMP [26] for in-ReRAM computing and Duality Cache [27] for in-SRAM computing. We use parameters from Ambit [59] for bit-serial in-DRAM computing. The execution trace from the autograd profiler is replayed in the simulator, and the actual input

Table III
MLIMP CONFIGURATIONS.

	Array				SIMD ALUs		MAC throughput / ALU		
	Dimension	# arrays	MB/mm^2	MHz	#ALUs/array	#ALUs	cycles/op (2ops)	MOPS (2ops)	MOPS (4ops)
SRAM	256 x 256	5,120	0.6	2,500	256	1.31 M	302	8.278	2.070
DRAM	8 KB x 8,192	1,024	17.5	300	65,536	67.1 M	1510	0.199	0.050
ReRAM	128 x 128 x 2 (bit/cell)	86,016	2.5	20	16	1.37 M	8	2.500	2.500

data is regenerated to perform the timing simulation in each module. Load and store bandwidth for the main memory communication is simulated using Ramulator [40] integrated into our simulator. The data transfer bandwidth between CPU and GPU for baseline GPU execution is recalculated using the actual bandwidth of the PCIe channels measured by CUDA Toolkit to bypass PyTorch's bottlenecks. Predictor latency is measured by a C++ implementation of the regressor models.

The power parameters for in-memory computing are taken from the prior work [26], [27], [59]. Power and energy for CPU and DRAM activity are measured by profiling microbenchmarks using Intel Rapl interface. We use NVIDIA nvprof to measure GPU power.

V. RESULTS

A. Configurations Studied

In this section, we evaluate the proposed MLIMP system. Our baseline is composed of a dual-socket Xeon E5-2697 v3 (64GB DDR4) server and NVIDIA Titan XP (12GB GDDR5) GPU. The system configuration for MLIMP is shown in Table III. We assume 336 MB ReRAM accelerator chip (scaled down from [26]). It has a similar area as the on-chip cache of a dual-socket CPU server. We use half of total SRAMs for in-cache computing allocation because reserving an SRAM portion for general caches is beneficial for both CPU processes and in-cache computing as suggested in [27]. In this configuration, SRAM and ReRAM have a similar number of SIMD ALUs. For in-DRAM computing, we assume DDR4-2400 memory with 4 channels, 1 rank, 16 chips, and 16 banks, supporting bank-level in-memory operations. Each baseline in-memory processor can handle up to 8 outstanding jobs at a time.

B. GNN Performance

1) Kernel Performance: In this section, we discuss the performance of kernels in GNN applications. We use the ogbl-citation2 dataset, but a similar trend is observed in most of the real-world graphs that we tested. The box chart of the distribution of the kernel speedup of MLIMP is shown in Figure 11. Compared to our baseline, we observe the average speedup of 4.07× for GEMM, 3.40× for SpMM, and 1.82× for Vadd. The massive parallelism of in-memory computing generally contributes to the speedup of compute-intensive kernels such as GEMM and Vadd. SpMM additionally benefits from the internal reuse of input features and input parallel execution.

The execution time breakdown for the three major kernels of our tested GNN, i.e. GEMM, SpMM, and vector add (Vadd), is shown in Figure 12. This assumes different combinations of in-memory devices are activated for acceleration. Compared with CPU, the compute kernels are significantly accelerated by GPU, while GPU execution incurs additional data transfer costs for transferring submatrices and input features to GPU. This data transfer is unavoidable especially when dealing with large graph data. In-memory computing can bypass the memcpy bottleneck by tight integration with the host memory hierarchy, although memory access time in each kernel sees a slight increase due to narrower DDR4 memory technology. From the different mixture of in-memory computing devices, we can see the SpMM kernel is dominating for all scenarios, while we see the smallest execution time in "SRAM and ReRAM" and "All".

Focusing on the execution time for SpMM, SRAM and ReRAM result in a similar kernel performance because they have a similar SIMD width and an average MAC throughput per SIMD slot considering the multi-operand operations. In-DRAM SpMM observes worse performance for SpMM due to the smaller SA density (Figure 1) and available array-level parallelism. While DRAMs have a large array width, their SIMD slots cannot be fully utilized by GNNs of a small feature vector size. Thus, although some subgraphs are mapped to DRAM in "All", it does not result in a noticeable speedup compared to "SRAM and ReRAM".

As discussed in detail in Section V-B3, this is 77% of the oracle case with a perfect job balancing across the memories.

2) Application Performance: Figure 13 shows the application time breakdown for different input graphs, normalized to the baseline GPU+CPU execution. *Others* include unparallelized pre- and post-processing time such as indexing, sigmoid, and prediction MLP. The pre-execution cost (e.g. predictor) takes an insignificant time ($< 2\%$ of SpMM kernel) even running on a single core.

We observe drastic speedup in the memcpy time and SpMM kernel for most of the input graphs. The speedup of GEMM is moderate compared to other kernels. This is because the majority of GEMM time is spent for data communication to fetch the input features, and we do not benefit from overlapped execution because the compute time is smaller than the data communication time due to the massively parallel execution. Interestingly, it is also observed that SpMM of some graphs performs poorly on SRAM (e.g., ogbl-ddi, ogbl-collab) or on ReRAM (e.g., ogbl-ppa). MLIMP

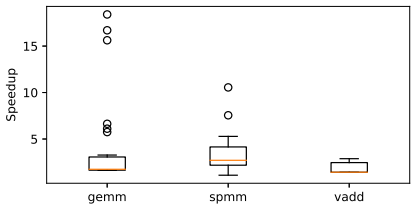


Figure 11. Kernel speedups (ogbl-citation2).

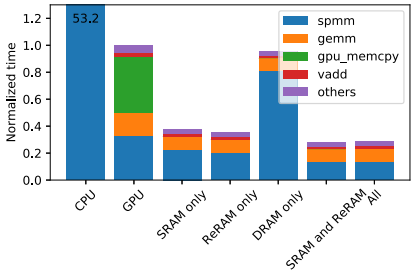


Figure 12. Kernel performance of different memories (ogbl-citation2).

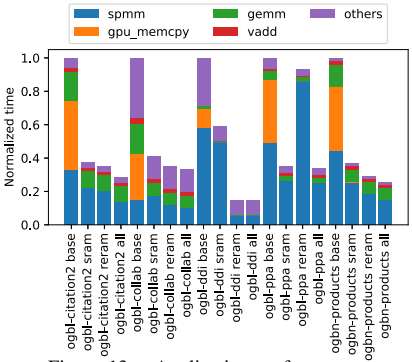


Figure 13. Application performance.

can pick the best memory and accelerate the execution, while offloading some jobs to suboptimal memory as well to increase the throughput. Overall we achieve 4.80× geomean speedup over GPU and 241× over CPU for the graphs we evaluated.

Figure 14 shows the energy consumption of the GNN applications. Because data transfer can take more energy than computation in the conventional CPU and GPU architecture, we achieve a greater energy benefit from in-memory computing which can reduce it. On average, we achieve 5.02× better energy efficiency for MLIMP over a GPU.

3) Scheduler and Predictor Performance: The performance of our job scheduler and performance predictor is illustrated in Figure 15. The results are based on the ogbl-citation2 dataset and use different job schedulers presented in Section III-E. We use an oracle predictor, which returns the accurate cycle counts of a job in each memory, and our MLP regressor based predictor. We compare the execution time for SpMM.

We notice that the local adaptive scheduler slightly decreases the performance compared to the global scheduler. This is mainly because of the bubbles caused by small fragmented resources that were not scheduled to any of the awaiting jobs in the queue. On the other hand, global scheduling results in the best performance, providing a highly balanced job schedule across different in-memory devices. We also notice our MLP regressor based performance predictor provides reasonably good performance estimates, and the scheduler performance gap between the oracle predictor and ours is trivial (less than 1%). The accuracy of the performance predictor also contributes to the global scheduler outperforming the others.

We conduct a stress test of the schedulers to measure the tolerance to impreciseness of the predictor with an artificial dataset that follows Pareto (scale-free) distribution. We observe the local adaptive scheduler results in better performance with added Gaussian noise of $\sigma > 0.39$ on average. In such a case, the global scheduler sees relatively large tail latency for the delayed job items, whereas the local adaptive scheduler can automatically adjust by itself which more than amortizes the bubble induced overhead. The error tolerance of the global scheduler becomes low if the batch size is small (threshold $\sigma = 0.25$ for a batch size of 16).

Figure 16 compares the performance of our approach with the oracle throughput, which assumes the perfect job balancing among the memories. The oracle performance is calculated by making a sum of the throughput of each in-memory processor. The baseline assumes the same server configuration as MLIMP, but uses naive LJF scheduling to schedule jobs. We observe that the scheduling approach of MLIMP achieves 77% of the best on average, while the naive baseline barely achieves 34%. It is notable that naive scheduling approach is likely to result in the single processor performance of the best in-memory processor, and further performance improvement can only be made by introducing an intelligent job scheduling approach.

C. Data-Parallel Applications

In this section, we evaluate the data parallel applications in MLIMP using several multiprogramming scenarios. We first present the kernel execution time of in-SRAM, in-DRAM, and in-ReRAM computing in Figure 17, normalized to the minimum of these three. In-DRAM computing assumes similar in-memory operations as in-SRAM computing, but it might not be the best design because no prior work has demonstrated general in-DRAM computing supporting non-trivial arithmetic operations with an execution model optimized for general data-parallel applications. The preference for in-memory devices depends on many factors as we discussed, while working data set size and instruction mix are some of the dominating ones. For example, DRAM prefers bulk bitwise operations but not complex operations, while SRAM can efficiently perform relatively small but compute-intensive kernels. ReRAM can efficiently run applications exploiting analog operation intrinsics such as dot product.

We then assume scenarios of launching multiple programs from the program set. All possible combinations of four applications are tested, and we pick combinations that show various in-memory device preferences as in Table II. For example, combination A is composed of applications that favor SRAM, while F favors DRAM+ReRAM. We compare the execution time in Figure 18. The execution time is normalized to MLIMP ALL. While the preferred set of memory depends on the type of programs launched, MLIMP

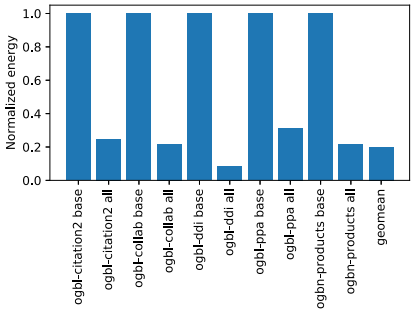


Figure 14. Application energy.

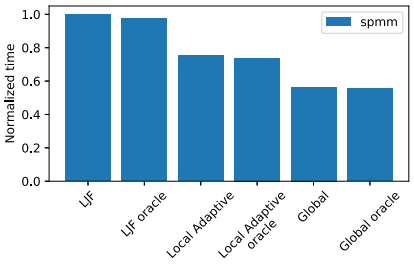


Figure 15. Scheduler performance (ogbl-citation2). Oracle uses the oracle predictor.

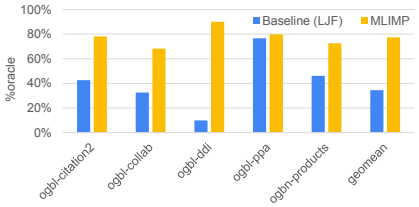


Figure 16. Performance compared to the oracle throughput with perfect job balancing.

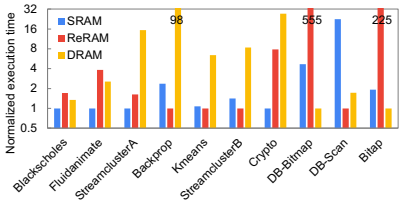


Figure 17. Single application.

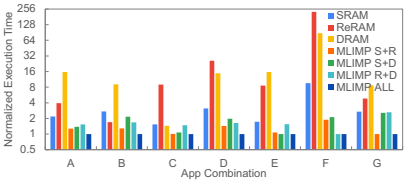


Figure 18. Multiple applications.

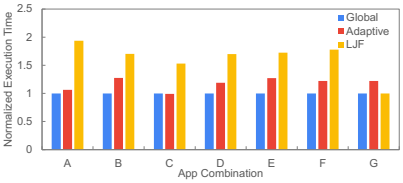


Figure 19. Scheduling comparison.

can schedule jobs to minimize the latency while balancing the load, resulting in the best performance. We also observe jobs from the same application can be scheduled to different memories for the sake of throughput. Compared to the runs on a single layer in-memory processing system, we achieve 7.1× better performance. Note that in-memory computing outperforms our GPU baseline for all applications we tested. Even when they run on a single type of memory, there are performance gaps of orders of magnitude (e.g. 15×). Therefore, for any app combinations, MLIMP outperforms our GPU baseline.

We also compare the performance of different scheduling approaches in Figure 19. Because the execution time of the compute kernels can be calculated deterministically, the global scheduler that can perform both local and global adjustment achieves the best performance for almost all scenarios.

We conclude that a system with MLIMP can benefit from the rich parallel in-memory processing resources and reduced data transfer. Moreover, good job scheduling and performance estimation allow such a system to exploit the heterogeneous characteristics of different in-memory devices for jobs with dynamism and substantial variation.

VI. Related Work

To the best of our knowledge, this is the first work demonstrating the feasibility of MLIMP and its advantage for data parallel application with workload dynamism. In this section, we discuss some of the closely related work.

The past decades have witnessed growing attention to near-memory computing, also termed processing in memory or PIM, and in-memory computing. They try to address the cost of moving data and the issues related to the memory wall. PIM solutions move compute near memory [4], [11], [22], [24], [30], [39], [52], [54], [56], [58], [66], [70] to reduce the data movement cost and utilize the bandwidth available by computing near memory. Specialized PIM architecture has been studied for various acceleration targets including machine learning [29], [31], [45], sparse data processing [29], [32], [37], and databases [20]. In contrast, in-memory computing architectures differentiate themselves by re-purposing memories themselves into compute engines, utilizing the physical properties of the memory array. This unlocks massive parallelism and reduced data movement, making them more efficient than PIM, at the cost of reduced flexibility for data alignment, etc.

Prior work has proposed instruction or thread scheduling schemes for PIM. PEI [5] uses a locality monitor to assess the performance of remote PIM execution, and AMS [64] leverages cache partitioning techniques to determine thread mappings for a system with multiple memory hierarchy depths. While their ultimate goal is to answer *whther or not* to offload for PIM, MLIMP is additionally required to determine resource allocation size and memory type, making the scheduling problem difficult. As we use a predictor for some classes of applications, increased fuzziness between memory and computing of IMP presents a new scheduling challenge.

Livia [48] proposes multi-layer *near-memory* computing using a common memory service unit implemented in each memory layer. It targets applications that couple short computation between irregular memory access, such as pointer chasing and tree traversal. It is pointed out that solely using PIM in the main memory leads to suboptimal performance because they cannot reap the benefit from the locality that caches can better harness. MLIMP has different application targets, i.e., applications with unpredictable reuse patterns and *compute intensity*. In addition to supporting various reuse patterns, in-memory computing offers massively parallel execution that is unseen in near-memory computing.

VII. Conclusion

We propose MLIMP that runs various computing kernels with workload dynamism in variable layers in the in-memory computing enabled memory hierarchy. By introducing a job scheduler and a performance predictor, GNN inference jobs, which show significant variation in the working dataset and reuse patterns, are well mapped to appropriate memories. Our multi-layer in-memory computing approaches provide performance advantages to multiprogramming scenarios for general data-parallel applications compiled for multiple in-memory device targets. Our experimental results show that MLIMP improves the performance of various GNN inference tasks in OGB by 4.80× over server class GPU, and general applications by 7.1× over single layer IMP. Re-purposing the existing memory hierarchy for multi-layer in-memory computing provides 5.02× better energy efficiency.

Acknowledgment

We thank the anonymous reviewers for their suggestions which helped improve this paper. This work was supported in part by the NSF under the CAREER-1652294 and NSF-1908601 awards, JSPS KAKENHI Grant Number JP22K21284, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] L. A. Adamic and E. Adar, "Friends and neighbors on the web," *Social Networks*, vol. 25, no. 3, pp. 211–230, 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378873303000091

[2] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 481–492.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 105–117.

[4] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, 2015.

[5] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 336–348.

[6] K. Aida, "Effect of job size characteristics on job scheduling performance," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1–17.

[7] Arm Ltd., "Arm big.LITTLE." [Online]. Available: https://www.arm.com/why-arm/technologies/big-little

[8] B. C. Arnold, "Pareto distribution," *Wiley StatsRef: Statistics Reference Online*, pp. 1–10, 2014.

[9] J.-P. Aumasson and D. J. Bernstein, "Siphash: a fast short-input prf," in *International Conference on Cryptology in India*. Springer, 2012, pp. 489–508.

[10] J. Bennett, S. Lanning *et al.*, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007. Citeseer, 2007, p. 35.

[11] J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge, "A low cost, multithreaded processing-in-memory system," in *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, ser. WMPI '04. New York, NY, USA: ACM, 2004, pp. 16–22. [Online]. Available: http://doi.acm.org/10.1145/1054943.1054946

[12] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[13] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[14] P. Chi, S. Li, and C. Xu, "PRIME : A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory," in *IEEE International Symposium on Computer Architecture*. IEEE, jun 2016, pp. 27–39. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7551380

[15] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 257–266.

[16] B. Y. Cho, J. Jung, and M. Erez, "Accelerating bandwidth-bound deep learning inference with main-memory accelerators," *CoRR*, vol. abs/2012.00158, 2020. [Online]. Available: https://arxiv.org/abs/2012.00158

[17] B. Y. Cho, Y. Kwon, S. Lym, and M. Erez, "Near data acceleration with concurrent host access," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 818–831.

[18] H. Cook, K. Asanovic, and D. A. Patterson, "Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments," Technical Report No. UCB/EECS-2009-131, Tech. Rep., 2009.

[19] I. Cutress, "Amd confirms milan-x with 768 mb l3 cache: Coming in q1 2022," Nov 2021. [Online]. Available: https://www.anandtech.com/show/17053/amd-confirms-milan-x-with-768-mb-l3-cache-coming-in-q1-2022

[20] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The mondrian data engine," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 639–651, 2017.

[21] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 383–396.

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

[23] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning, "In-memory intelligence," *IEEE Micro*, vol. 37, no. 4, pp. 30–38, 2017.

[24] B. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas, "Programming the flexram parallel intelligent memory system," *SIGPLAN Not.*, vol. 38, no. 10, pp. 49–60, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/966049.781505

[25] D. Fujiki, N. Chatterjee, D. Lee, and M. O'Connor, "Near-memory data transformation for efficient sparse matrix multi-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–17.

[26] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173171

[27] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 397–410.

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

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

[30] M. Gao and C. Kozyrakis, "Hrl: Efficient and flexible reconfigurable logic for near-data processing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Ieee, 2016, pp. 126–137.

[31] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 751–764.

[32] C. Giannoula, I. Fernandez, J. G. Luna, N. Koziris, G. Goumas, and O. Mutlu, "Sparsep: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 1–49, 2022.

[33] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *arXiv preprint arXiv:1709.05584*, 2017.

[34] C. Hong, A. Sukumaran-Rajam, B. Bandyopadhyay, J. Kim, S. E. Kurt, I. Nisa, S. Sabhlok, U. V. Çatalyürek, S. Parthasarathy, and P. Sadayappan, "Efficient sparse-matrix multi-vector product on gpus," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 66–79. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/3208040.3208062

[35] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint arXiv:2005.00687*, 2020.

[36] S. M. Johnson, "Optimal two- and three-stage production schedules with setup times included," *Naval Research Logistics Quarterly*, vol. 1, no. 1, pp. 61–68, 1954. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800010110

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

[38] B. Keeth, R. J. Baker, B. Johnson, and F. Lin, *DRAM circuit design: fundamental and high-speed topics*. John Wiley & Sons, 2007, vol. 13.

[39] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Proceedings of ISCA*, vol. 43, 2016.

[40] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator." *Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[41] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[42] R. Kolisch, "Efficient priority rules for the resource-constrained project scheduling problem," *Journal of Operations Management*, vol. 14, no. 3, pp. 179–192, 1996.

[43] R. Kolisch and S. Hartmann, *Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis*. Boston, MA: Springer US, 1999, pp. 147–178. [Online]. Available: https://doi.org/10.1007/978-1-4615-5533-9_7

[44] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve, "Stash: Have Your Scratchpad and Cache It Too," in *ISCA'15*, 2015, pp. 707–719. [Online]. Available: http://rsim.cs.illinois.edu/Pubs/15-ISCA-stash.pdfhttp://dl.acm.org/citation.cfm?id=2749469.2750374

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

[46] S. Li, A. O. Glova, X. Hu, P. Gu, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Scope: A stochastic computing engine for dram-based in-situ accelerator." in *MICRO*, 2018, pp. 696–709.

[47] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 288–301.

[48] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, "Livia: Data-centric computing throughout the memory hierarchy," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 417–433. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/3373376.3378497

[49] A. Mehrabi, D. Lee, N. Chatterjee, D. J. Sorin, B. C. Lee, and M. O'Connor, "Learning sparse matrix row permutations for efficient spmm on gpu architectures," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 48–58.

[50] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *2017 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 2017, pp. 457–468.

[51] K. S. Naphade, S. D. Wu, and R. H. Storer, "Problem space search algorithms for resource-constrained project scheduling," *Annals of operations research*, vol. 70, pp. 307–326, 1997.

[52] M. Oskin, F. T. Chong, T. Sherwood, M. Oskin, F. T. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory," *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 192–203, 1998.

[53] L. Özdamar and G. Ulusoy, "A survey on the resource-constrained project scheduling problem," *IIE transactions*, vol. 27, no. 5, pp. 574–586, 1995.

[54] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *Micro, IEEE*, 1997.

[55] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for Cross-CPU attacks," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 565–581. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[56] S. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014.

[57] Y. Qi, Z. Bar-Joseph, and J. Klein-Seetharaman, "Evaluation of different biological data and computational classification methods for use in protein interaction prediction," *Proteins: Structure, Function, and Bioinformatics*, vol. 63, no. 3, pp. 490–500, 2006.

[58] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46.

[59] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 273–287.

[60] A. Shafiee, A. Nag, N. Muralimanohar, and R. Balasubramonian, "ISAAC : A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 14–26, jun 2016. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7551379

[61] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," in *Proceedings - International Symposium on High-Performance Computer Architecture*, 2017, pp. 541–552. [Online]. Available: http://alchem.usc.edu/portal/static/download/nn{_}memristor.pdf

[62] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 531–543.

[63] Z. Stanfield, M. Coşkun, and M. Koyutürk, "Drug response prediction as a link prediction problem," *Scientific reports*, vol. 7, no. 1, pp. 1–13, 2017.

[64] P.-A. Tsai, C. Chen, and D. Sanchez, "Adaptive scheduling for systems with asymmetric memory hierarchies," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 641–654.

[65] X. Xin, Y. Zhang, and J. Yang, "Roc: Dram-based processing with reduced operation cycles," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[66] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "Top-pim: Throughput-oriented programmable processing in memory," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14, 2014.

[67] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 544–557.

[68] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," *Advances in Neural Information Processing Systems*, vol. 31, pp. 5165–5175, 2018.

[69] M. Zhang, P. Li, Y. Xia, K. Wang, and L. Jin, "Revisiting graph neural networks for link prediction," *arXiv preprint arXiv:2010.16103*, 2020.

[70] Q. Zhu, B. Akin, H. Sumbul, F. Sadi, J. Hoe, L. Pileggi, and F. Franchetti, "A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing," in *3D Systems Integration Conference (3DIC), 2013 IEEE International*, 2013.