DP-Sim: A Full-stack Simulation Infrastructure for Digital Processing In-Memory Architectures

Minxuan Zhou, Mohsen Imani*, Yeseong Kim** , Saransh Gupta, and Tajana Rosing
Department of Computer Science and Engineering, UC San Diego, La Jolla, USA
*Department of Computer Science, UC Irvine, Irvine, USA
**Department of Information and Communication Engineering, DGIST, Daegu, Republic of Korea
{miz087, sgupta, tajana}@ucsd.edu, m.imani@uci.edu, yeseongkim@dgist.ac.kr

ABSTRACT

Digital processing in-memory (DPIM) is a promising technology that significantly reduces data movements while providing high parallelism. In this work, we design and implement the first full-stack DPIM simulation infrastructure, DP-Sim, which evaluates a comprehensive range of DPIM-specific design space with respect to both software and hardware. DP-Sim provides a C++ library to enable DPIM acceleration in general programs while supporting several aspects of software-level exploration by a convenient interface. The DP-Sim software front-end generates specialized instructions that can be processed by a hardware simulator based on a new DPIM-enabled architecture model which is 10.3% faster than conventional memory simulation models. We use DP-Sim to explore the DPIM-specific design space of acceleration for various emerging applications. Our experiments show that bank-level control is 11.3× faster than conventional channel-level control because of higher computing parallelism. Furthermore, cost-aware memory allocation can provide at least 2.2× speedup vs. heuristic methods, showing the importance of data layout in DPIM acceleration.

1 INTRODUCTION

The "memory wall" issue has become a major bottleneck in conventional Von Neumann architectures. One promising solution is Processing in-memory (PIM), which offloads computations to memory, significantly reducing data movement while providing high parallelism. There have been many PIM designs based on different memory technologies. Some of them exploit the analog computing of Non-Volatile Memories (NVMs) to accelerate various applications, including machine learning [21], graph processing [22], and high-performance computing [5]. However, these accelerators have several critical limitations, including costly peripherals for data conversion, no floating-point support, and scaling difficulty due to unstable multi-bit cells [8, 21]. As reported in previous work, analog-to-digital (ADCs) and digital-to-analog converters (DACs) consume over 83.3% power in the analog PIM (APIM) DNN accelerator [21].

In addition to analog PIM, we can also enable the PIM functionality in the digital memory, called digital PIM (DPIM), using conventional memory technologies including SRAM [1, 4], DRAM [6, 15] and ReRAM [8]. Figure 1 shows an example of ReRAM-based DPIM block. The basic DPIM operations are bit-wise operations (e.g. full addition) between multiple memory rows which can implement element-wise operations on binary vectors (Figure 1(b)). We can use multiple serial steps of single-bit operations to enable element-wise computation for multi-bit vectors by allocating multiple rows to each vector. Using bit-serial row-parallel operations, a DPIM memory block can act like a SIMD processing unit that processes multiple computations simultaneously. DPIM supports arithmetic operations (e.g. addition, multiplication, division, etc.) as well as in-memory search operations (Figure 1(c)). Considering the large

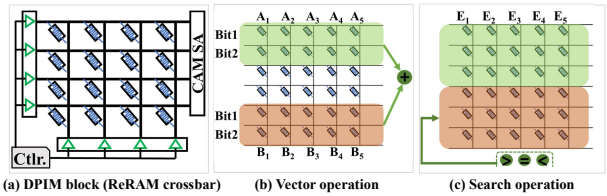


Figure 1: An example of a ReRAM-based digital PIM.

number of memory blocks in a memory system, DPIM provides extremely high-degree parallelism. At the same time, unlike analog PIM, DPIM operates on binary (digital) values, avoiding inefficient conversion peripherals like ADCs and DACs. Because of these advantages, researchers have been actively working on the chip fabrication [9, 23] which shows promising results.

To efficiently and comprehensively evaluate different DPIM designs, architectural simulation is important. However, we cannot easily simulate DPIM-enabled systems using existing tools for two reasons. First, the efficiency of DPIM acceleration strongly depends on the layout of application data in the memory because of the bit-serial row-parallel operations. DPIM system needs to manage DPIM data in a different way from that used in conventional computing systems (e.g. *malloc*() function). To the best of our knowledge, there is no simulation tool providing the software interface for implementing and exploring the DPIM-specific layout for general applications. Second, traditional memory simulators such as Ramulator, DRAMSim2, and NVMain [10, 19, 20] cannot support exploration of DPIM architectures because the base architecture model is designed to support only conventional memory. To explore DPIM-specific designs in existing tools, we have to use several tools and heavily modify both software-level interface and hardware model, which are both time-consuming and challenging.

In this work, we propose and implement a full-stack simulation infrastructure, DP-Sim, for exploring design space of DPIM architectures from both software and hardware perspectives. DP-Sim consists of two main components: the front-end for software implementation and the back-end for hardware simulation. The DP-Sim front-end contains a DPIM library and a configurable compiler layer to generate instruction trace of application based on user-defined software-level design. We can use the DPIM library to declared DPIM-specific data structures and functions in application source codes, and design data layout strategies by using the interface of a configurable data allocator. The DP-Sim front-end then compiles DPIM application to generate instructions which can be simulated by DP-Sim back-end. The back-end of DP-Sim is a cycle-accurate memory simulator for DPIM-enabled architectures. The DP-Sim back-end utilizes a new memory architecture model which supports exploration on several DPIM-specific designs like PIM instruction scheduling and operation parallelism. We propose a fast-forward synchronous model for DPIM instructions to improve the speed of simulation. To the best of our knowledge, DP-Sim is the first simulation infrastructure to support systematic evaluation for general applications running on DPIM architectures.

ASP-DAC '21, Jan.18-21, 2021, Tokyo, Japan

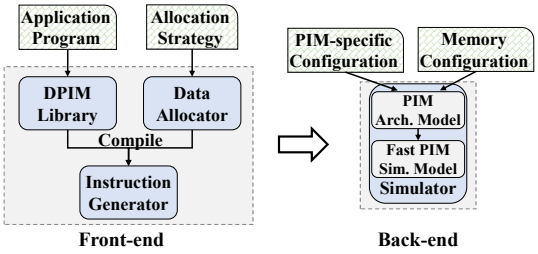


Figure 2: The overview of DP-Sim infrastructure.

```
DpData v1 = dp_data(32, ["int32"]),
       v2 = dp_data(32, ["int32"]),
       v3 = dp_data(32, ["int32"]); // Declare an int32 vector
DpData tab = dp_data(32, ["int32","int16","bool"]); // Declare a table with 3 fields
decl_comp("add", v1, v2, v3); // Declare a vector computation
dp_alloc([v1,v2,v3,tab]); // Allocate memory
dp_vec_add(v1, v2, v3); // DPIM vector operation
dp_exact_search(tab, "0=123"); // DPIM table search -> field0 = 123
```

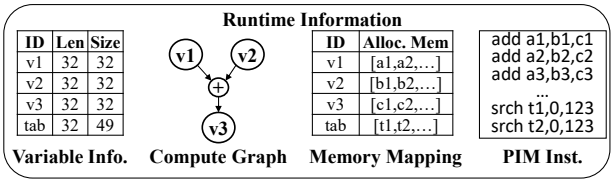


Figure 3: The high-level design of the DPIM library.

We evaluate the design of DP-Sim by running various emerging data-intensive workloads from fields of graph processing, database, machine learning, and data mining. We compare the simulation of DP-Sim with a PIM-enabled version of Ramulator [10]. The experiment shows that DP-Sim is 10.3× faster with only 6.3% difference in the result. We use DP-Sim to explore several DPIM-specific design choices including architecture hierarchy, control granularity, and adaptive data layout. Our experiments find that a bank-level control is 11.3× faster than conventional channel-level control because of higher parallelism. Furthermore, a cost-aware memory allocation provides a 2.2× speedup over heuristic methods, indicating the importance of data layout in DPIM acceleration.

2 DP-SIM DESIGN

Figure 2 shows the high-level view of DP-Sim, consisting of the software interface (front-end) and the architectural simulation (back-end). The DP-Sim front-end provides the interface for implementing DPIM-accelerated programs through a DPIM library and a configurable data allocator. Specifically, the library provides function calls to implement DPIM data structures and operations while the data allocator determines the memory layout for application data based on the user-defined allocation strategy. The front-end generates DPIM-accelerated instructions for hardware simulation, which is processed by the DP-Sim back-end architectural simulators to evaluate the design. The hardware simulator utilizes a new DPIM-enabled hardware model to support efficient exploration on DPIM-specific design. The rest of this section introduces details of software front-end and hardware back-end.

2.1 DP-Sim Front-end

The DP-Sim front-end provides the interface of implementing DPIM-accelerated applications and generates the DPIM-enabled instructions through three key components: DPIM library, data allocator, and instruction generator.

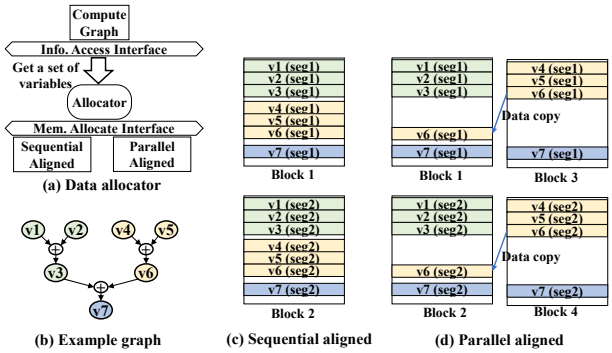


Figure 4: The impact of different allocation mechanisms.

2.1.1 DPIM Library. The DPIM library contains three categories of functions for declaring DPIM data, allocating memory, and executing DPIM operations respectively. Figure 3 shows an example of utilizing these functions in the application source code.

DPIM Data Structure. To utilize the DPIM functionality, programs need to declare application variables as specific DPIM data structures. The DPIM library provides declaration functions for two basic DPIM data structures, vector and table, which can be represented by a general data structure, *DpData*. To declare a *DpData* variable, the programmer needs to declare the data dimension including the number of elements and a vector representing the data type of each element. For a DPIM vector, the vector has one specific type (e.g. integer, fixed-point, floating-point, etc.); for a DPIM table, each element may have multiple types for different fields in the table.

Data and Dependency Declaration. When calling declaration functions, the DPIM library only records the information of declared variables without allocating memory for variables. Such a design enables us to explore strategies that optimize the data layout of multiple DPIM variables. In addition to declaration functions for variables, the DPIM library provides functions for declaring DPIM operations to record the dependency between variables, which is helpful design an efficient memory layout (Section 2.1.2). As shown in Figure 3, we can build a compute graph with one "add" node based on three vector declarations and one computation declaration by a *decl_comp*("*add*") function.

Memory Allocation. Before issuing operations for declared DPIM variables, the programmer needs to call a *dp_alloc*() function to allocate memory space for these variables. The *dp_alloc*() function takes in a list of variables and allocates memory for input variables by a configurable data allocator. The library records the mapping information of DPIM variables in a table as shown in Figure 3. For each variable, the memory allocation is represented as a list of addresses of memory segments. We introduce details of the data allocator and allocation of memory segments in Section 2.1.2.

DPIM Operations. After allocating memory for DPIM variables, the programmer can process these variables by calling functions for DPIM operations. The DPIM library includes functions for all supported operations taking in both vectors and tables. These DPIM functions can generate DPIM instructions by extracting operand addresses from the memory mapping table. Section 2.1.3 introduces how DP-Sim generates PIM-enabled memory instructions by an instruction generator.

2.1.2 Data Allocator. When the program calls a *dp_alloc*() function, the DP-Sim front-end allocates memory for DPIM variables through a configurable data allocator (DA) which provide flexible interface for accessing software-hardware information and customizing the allocation strategy, as shown in Figure 4.

Table 1: DPIM Instruction Format.

	Vector operations	Search Operations
OpCode	[add, mul, bit-wise...]	[eq_s, min_s, max_s]
Operand1	seg_addr (source1)	mem_addr (start address)
Operand2	seg_addr (source2)	value (target value)
Operand3	seg_addr (destination)	type ([int32/int16/float16...])
Operand4	type ([int32/int16/float16...])	null

Accessing Information. To design an efficient data allocation strategy, we may need to consider various types of application information including size of each variable and compute dependency graph (bottom part of Figure 4). DA automatically generates the application information based on function calls in DPIM library. Researchers can use the interface of accessing such information when designing customized data layout. Furthermore, DA provides the interface to access hardware configurations which also affects the detailed data layout.

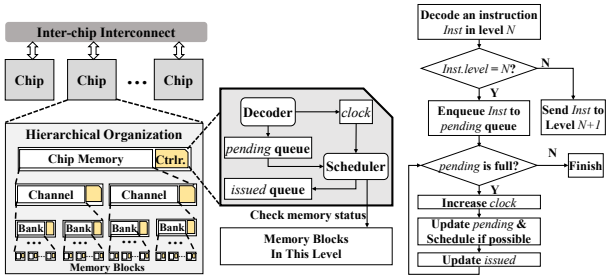
Designing Layout Strategy. DA provides a flexible interface for designing allocation strategy based on two basic allocation schemes. Figure 4 shows an example of the memory allocation for processing three vector computations using these two basic schemes. The sequential method aligns all vectors of these two computations in the memory block sequentially and allocates more blocks for vectors whose lengths exceed the limit of bit-lines. This maximizes the memory utilization and avoids data movement if two computations are dependent on each other. However, memory blocks storing two computations need to process them sequentially.

The parallel aligned method places vectors of two computations to different blocks where all computations can be processed in parallel. Though the parallel aligned method improves the compute performance, we need to move the data between different blocks if two results involve in a following computation. Such costs of computation and data movement vary as a function of both software and hardware. Therefore, an efficient allocation may adaptively select schemes for different DPIM variables.

Figure 4(a) shows the process of customizing the allocation strategy in DP-Sim. Specifically, we can traverse the compute graph and selects a set of vertices based on the user-define analysis of application information. For each set of vertices, we can exploit either the sequential scheme or the parallel scheme to allocate memory. In Section 3.5, we implement a cost-aware allocation method which provides significant speedup over fixed strategies.

Memory Segment Allocation. To manage the memory mapping information, we adopt a segmentation method where each variable is mapped to a set of memory segments. Each segment indicates a set of continuous rows in a specific memory block. The number of segments for a specific data structure depends on the number of elements in the variable as well as the memory technology. As shown in Figure 4, each vector needs two segments to store all values. When calling a *dp_alloc* function for a set of variables, DA automatically generates numbers of blocks and segments required for each allocation scheme based on software/hardware information. We can use this information to select memory blocks for segment allocation. With such segment allocation, DP-Sim is flexible to generate instructions on different DPIM technologies because memory segment is the basic unit in most DPIM architectures.

2.1.3 Instruction Generator. The DP-Sim front-end generates instructions for DPIM program by an instruction generator (IG). IG generates all memory-related instructions, including normal memory operations and PIM operations. We implement IG as a binary instrumentation tool based on Intel Pin [16], which captures functions from the DPIM library and other memory operations by actually running the application. The user-defined data allocator is a part of IG that runs during the instrumentation. IG evaluates the full memory performance of DPIM systems and can be integrated



(a) The overview of architecture model. (b) The logic of each controller.

Figure 5: The overview of architecture used in DP-Sim.

in other Pin-based CPU architecture simulators (e.g. Sniper [3]) for full-system simulation by replacing the memory model.

As mentioned before, each DPIM variable may occupy several memory segments. Therefore, we adopt a "single-segment" design for PIM instructions where each instruction indicates an operation in a memory segment. This design enables fine-grained control and scheduling mechanisms in the hardware side, which can provide more exploration flexibility. For example, to generate "single-segment" instructions for a DPIM vector operation, IG generates one DPIM instruction for each group of memory segments storing elements from the operands and the result. As shown in Figure 4, all vectors ($v1 - v7$) have two memory segments and we generate two single-segment instructions for each computation, where each instruction processes aligned segments of operands.

Table 1 lists formats of two types of DPIM operation. For vector operations, each instruction requires 3 operands including memory segment addresses for source and destination, as well as the data type. For each search instruction, we need to specify the the start memory address, the target value, and the type of target value. IG can then generate instructions with appropriate operands based on information in data allocator.

2.2 DP-Sim Back-end

DP-Sim back-end takes in the instruction trace generated by the front-end to evaluate the detailed system performance for a specific architecture configuration.

2.2.1 Architecture Model. Figure 5(a) shows the architecture model used in the DP-Sim back-end. Similar to widely-used memory simulators [10, 19, 20], we adopt a hierarchical model to provide the flexibility for designing various memory architectures. Figure 5(a) shows an example of three-level hierarchy: chip, channel, and bank. The highest level in the hierarchy is the memory chip where each chip connects to general data transfer interface to communicate with the host or other chips. This design can simulate multi-chip or multi-tile architecture which is widely used in emerging PIM accelerators [8, 21].

Inside each chip, we can configure the hierarchy for a specific memory organization. Each component at each level contains a memory partition and a controller that can process operations in the memory partition. Each component may have multiple sub-components at the lower level where each sub-component handles a subset of the memory. The basic unit in the architecture is the memory block that has an array of memory cells and peripheral circuits to support PIM operations.

Conventional memory designs usually place controllers at a high level in the hierarchy (e.g. channel) because the memory performance is bounded by the bandwidth of transferring data to the host. In PIM-enabled memory, however, PIM operations in different blocks do not share the global bus so that the fine-grained control

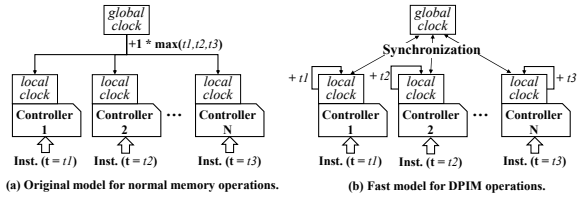


Figure 6: The comparison between simulation models used for a) normal memory operations, and b) DPIM operations.

scheme may increase the system performance. The DP-Sim back-end enables us to configure controllers at all levels in the memory hierarchy where controllers at each level can process a specific set of operations. In the next section, we introduce the detailed design of the PIM-enabled controller.

2.2.2 PIM-Enabled Controller. Figure 5(a) shows several key components in each memory controller for processing memory instructions and Figure 5(b) shows the detailed logic of each controller. Each memory controller maintains a `clock` to control the timing for all operation in a cycle-accurate way. When receiving an instruction, a decoder captures the detailed information of the instruction including the operation type and memory addresses. As mentioned previously, DPIM-enabled memory, unlike conventional memory, may have controllers for different levels in the memory hierarchy. Therefore, the decoder checks at which level the instruction needs to be processed. If the instruction needs to be processed in a lower level, the controller sends it to a corresponding sub-component based on the memory addresses.

If the controller needs to process an instruction, it adds the instruction to a queue which stores all instructions currently waiting to be scheduled. A scheduler checks the `pending` queue at each clock tick and schedules the next instruction if memory components can process the corresponding operations. The controller adds each issued instruction to the `issued` queue which stores all instructions that has been issued while have not completed. The controller checks the `issued` queue at every clock tick and updates the memory status if an instruction completes.

Since each instruction follows the "single-segment" format, the hierarchical PIM-enabled controller design enables us to implement a fine-grained control scheme, where each block can independently process instructions. For PIM operations involving a large number of memory blocks, the conventional channel-level control scheme may not be the best solution. In this case, the fine-grained control can significantly improve the system throughput because all memory blocks can work simultaneously.

2.2.3 DPIM Simulation Acceleration. Conventional memory simulators usually adopt a cycle-accurate model which is similar to the logic shown in Figure 5(b). When processing each instruction, this cycle-accurate model increases clocks of all memory controllers to maintain a global clocks, as shown in Figure 6(a). This model can simulate fine-grained behaviour accurately so we use it for simulating normal memory operations. However, the original cycle-accurate model may become extremely slow for DPIM simulation because a DPIM operation may introduce a large amount of "single-segment" instructions. To improve the simulation speed, we propose a fast simulation model for efficiently simulating DPIM operations without significantly losing the accuracy.

Figure 6(b) shows the proposed DPIM simulation acceleration as compared to the original cycle-accurate model (Figure 6(a)). Unlike the original model, the fast PIM model only advances local clock where different "single-segment" instructions may come from a single parallel operation (e.g. vector operation). Since normal memory operations may change the global clock, we clear all pending queues for normal memory operations if the simulator needs to schedule PIM instructions. Considering each PIM instruction takes many more cycles than normal memory operations (e.g. 100 cycles for a 8-bit addition v.s. 1 cycle for a normal read), we use a fast-forward model to simulate PIM instructions. We directly add the latency to local clock of the controller handling executing the instruction, saving the time for cycle-by-cycle simulation. After scheduling all DPIM instructions in the pending queue, we update the global clock by synchronizing local clocks in all memory controllers.

This fast-forward method would not significantly effect the simulation accuracy because a DPIM operation (a function call from the application) usually requires a large amount of "single-segment" instructions which keeps the simulator in the DPIM mode for a long enough time. Furthermore, each controller in the lowest level still handles only one PIM instruction at the same time, not violating any hardware constraint. Our experiment shows that the proposed model can significantly speed up the detailed cycle-accurate model while providing similar simulation results 3.

2.2.4 PIM Instruction Emulation. Considering there are various DPIM memory technologies, including SRAM, DRAM, and non-volatile memories, DP-Sim provides the interface for researchers to design customized instructions with specific timing and energy parameters. Architecture researchers can input validated parameters from low-level simulation tools like Cacti [18] or HSPICE. DP-Sim back-end maintains the constraint that each memory block can only process one operation each time and periodically updates block information based on the latency of scheduled operations. Such scheme is general to high-level simulation for operations based on different technologies.

3 RESULTS

We implement all components of DP-Sim in C++. The DP-Sim front-end contains a DPIM-specific library and an instrumentation tool based on Intel Pin-tool [16] for data allocation and instruction generation. To customize the data layout, we provide a base class for user to implement specific strategy by utilizing the interface introduced in Section 2.1. The DP-Sim back-end is a modified version of Ramulator [10], which is a validated and widely-used cycle-accurate memory simulator. We provide the file configuration interface for customizing the simulated architecture.

In this work, we focus on resistive memory based PIM technology, which has been extensively used in a wide range of accelerators [5, 8, 21, 22]. We should note that DP-Sim also supports other memory technologies and we show a case study of different memory technologies in Section 3.5. The basic NVM technology used in this work is the Voltage ThrEshold Adaptive Memristor (VTEAM) model [13] with I_{ON}/I_{OFF} ratio of 10^3. We use HSPICE design tool for circuit-level simulations, which provide timing and energy results for various ReRAM operations. We also exploit several existing tools, including Cacti [18], McPAT [14], and published data [17], to simulate timing and energy consumption of two other DPIM technologies: SRAM [1] and DRAM [6]. Our experiments do not include hardware validation for specific memory technology because we customize instructions based on parameters validated from other tools.

Because of the page limitation, our evaluation uses a fixed memory hierarchy, which is a three-level chip-channel-bank hierarchical architecture. Each bank is 8MB, consisting of 64 memory blocks, each of which has 1K bit-lines and 1K word-lines. Each channel can independently handle data transfers with a 128-bit link introducing 1 cycle for wire and the zero-load delay; for DPIM systems with

Table 2: Comparison with Ramulator-based model [10].

	Runtime (min.)		Cycles (10^6)		
Workload	Ramulator	DP-Sim	Ramulator	DP-Sim	Diff.(%)
bfs	62.1	9.3	59.8	64.1	7.2
sssp	100.3	10.4	80.6	71.4	-11.4
pr	45.3	7.9	29.3	28.7	-1.9
hash	14.5	1.3	14.9	15.1	1.1
query	20.3	2.1	13.4	14.6	9.3
alexnet	353.5	20.2	299.7	305.6	2.0
kmeans	10.4	0.9	8.1	7.3	-10.9

multiple chips, the inter-chip network is modeled as SerDes link used by HMC with an average 160GB/s bandwidth [7].

3.1 Applications and Workloads

To comprehensively test different designs, we implement several emerging data-intensive applications in DP-Sim.

Graph Processing. We implement three popular graph kernels including breadth-first search (BFS), single-source shortest path (SSSP), and page rank (PR). Graph information, including edge and vertex, is stored as tables that support in-memory searches. We transform vertex-related operations to DPIM vector operations to exploit the parallelism. We run these graph kernels with synthetic graphs with 1M vertices generated by GAP benchmarks [2].

Database. We implement two widely-used database kernels, hash table join (HASH) and query (QR), to process a large table using PIM search operations. Each table has 10M entries and each entry takes 1024 bits.

Machine Learning. We implement a popular deep neural network model, AlexNet [12], by transforming convolution and fully-connected layers to DPIM vector operations. We test inference with 8-bit prevision which has been proved to provide reasonable accuracy in recent works [11].

Data Mining. We implement K-Means (KM) clustering algorithm using DPIM acceleration. KM clusters high-dimensional data points by exploiting in-memory search and subtractions to update centers at each iteration.

3.2 Validation of DP-Sim

Considering the lack of publicly available PIM hardware, we validate our hardware simulation against a PIM-enabled Ramulator model [10]. Even though the original Ramulator is validated against Micron's DDR3 Verilog model, its simulation model can be extended to support various memory technologies with appropriate timing parameters [10]. Therefore, we implement different PIM operations as special memory write commands by customizing state machines and timing constraints with parameters validated from other low-level tools. The PIM-enabled Ramulator simulates fine-grained behaviors of PIM instructions in a cycle accurate way. We should note that this work focuses on the efficiency of software-hardware co-design for DPIM architectures while detailed circuit-level simulation is separately done.

We configure both tools to simulate a 8Gb memory chip wit 8-channels for ReRAM-based PIM architectures. We generate the same instruction trace using the DP-Sim front-end with a parallel aligned allocator (Section 2.1.2). For both tools, all instructions are processed by channel-level controllers. We compare the simulation time and performance difference between DP-Sim and Ramulator and show the results in Table 2. Our experiments show that DP-Sim is 10.3 × faster than Ramulator while exhibiting only 6.3% difference in the simulation results on average. Based on the results, DP-Sim can significantly improve the simulation speed of PIM acceleration using the proposed model with a sufficient accuracy. Furthermore, DP-Sim enables computer architects to explore a large design space for PIM architectures. We show the results of full-stack exploration in the following sections.

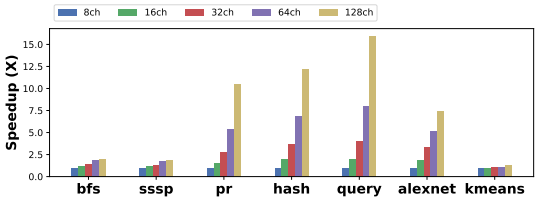


Figure 7: Performance of systems with different number of channels in a 8Gb memory chip.

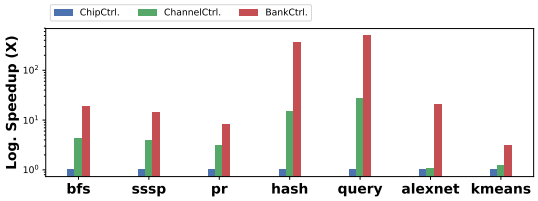


Figure 8: Performance of different controller granularities.

3.3 DPIM Parallelism Exploration

We utilize DP-Sim to explore the impact of architectural parallelism on the performance of DPIM system. In these experiments, we adopt the parallel aligned allocation discussed in Section 2.1 to schedule as many independent computations in parallel as possible.

Hierarchy Structure. We explore a DPIM system with per-channel PIM controllers to investigate the impact of the number of independent memory components. In such systems, the computing parallelism is determined by the number of channels. We scale the number of channels while keeping the memory capacity of each chip constant at 8Gb by decreasing the number of banks in each channel. Figure 7 shows the speedup of systems over the baseline system with 8 channels per chip. Based on the results, more channels, which provide a higher degree of parallelism, can improve the performance of most applications except `kmeans`, which is only 1.3× faster with 16× more channels. The reason behind this is that `kmeans` has a large number of data movements. The performance improvement provided by more channels depends on the maximum parallelism existing in applications (e.g. `bfs` v.s. `pr`). Such experiments show that more independent memory components can significantly improve the performance of highly parallel applications.

Controller Placement. In the above experiment, the parallelism is limited by the number of channels in the system. To further increase the parallelism, we utilize DP-Sim to explore different granularities of controller placement in DPIM architectures. In addition to the channel-level scheme, which is used in conventional memory systems, we test the bank-level and the chip-level schemes in a 8Gb per chip baseline system where each chip has 32 channels. Figure 8 shows normalized results to chip-level control design for each workload. The results show that bank-level control can provide 52.1× and 11.3× performance improvements over chip-level and channel-level schemes, because of the significantly increased parallelism. This experiment show the importance of processing DPIM instructions in a more fine-grained way than normal channel-level memory operations.

3.4 Memory Technology Exploration

We can also utilize DP-Sim to investigate the impact of memory technologies for DPIM acceleration. We simulate a state-of-the-art

Table 3: Comparison of different DPIM DNN accelerators.

Tech.	SRAM [4]		DRAM [15]		ReRAM [8]	
Size (MB)	Time(ms)	Energy (J)	Time(ms)	Energy (J)	Time(ms)	Energy (J)
8	10.21	2.82	65.72	0.48	175.37	0.12
16	8.31	3.23	33.25	0.63	88.92	0.22
32	7.56	4.11	17.14	1.00	46.26	0.31
64	6.98	6.20	11.83	1.82	25.52	0.49

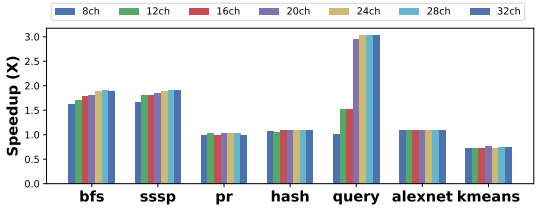


Figure 9: Performance improvement of parallel v.s. sequential method.

DPIM DNN accelerator [4] by DP-Sim and test different memory sizes and technologies. In addition to SRAM, which is used in the original paper, we test DRAM and ReRAM because both of them have been used in recent DPIM-based DNN accelerations [8, 15]. We simulate the inference task of AlexNet [12] with 8-bit precision. Since the original SRAM-based accelerator needs to load data from DRAM for each DNN layer, we model it as a two-chip system and configure the inter-chip speed similar to memory-cache data transfers. For the DRAM-based and ReRAM-based accelerators, we load the data from the same chip to simulate the separate memory partition for storing weights. We scale the memory size and compare the performance and energy consumption of different accelerators, as shown in Table 3. The results show that larger memory can significantly improve both DRAM (5.6×) and ReRAM (6.9×), but not SRAM (1.5×) whose performance is bounded by data transfer between heterogeneous memory components. Furthermore, the performance difference between different technologies becomes smaller when the memory size increases. We need to consider both application and hardware information when designing efficient DPIM acceleration.

3.5 Data Layout Exploration

With the help of DP-Sim, we can evaluate different data allocation methods with various hardware configurations.

Sequential v.s. Parallel Allocation. We first compare performance of sequential and parallel aligned allocation schemes, discussed in Section 2.1.2, for all tested applications in a single-chip system with various number of channels. Similar to previous experiments, each channel has 16 memory banks. As shown in Figure 9, the parallel aligned allocation can significantly improve the performance of several applications, especially `bfs` (1.9×), `sssp`(1.9×), and `query`(3.1×), because these applications contains many independent computations that can be processed in parallel by aligning them in different blocks. Furthermore, the speedup provided by the parallel allocation increases when we use more channels to provide more parallelism.

Cost-aware Data Allocation. We then propose and implement a new data allocation method which selects different schemes for different DPIM data structures based on the software and hardware information. The proposed algorithm considers the trade-offs between compute parallelism and data movement overhead of two basic allocation schemes. The proposed algorithm first breaks the compute graph into multiple sub-graphs of independent computations and then determines the allocation method for each sub-graph based on the estimated costs of computation and data movements. If the estimated cost of sequential aligned method for a sub-graph is higher than that of parallel aligned method, we sequentially align

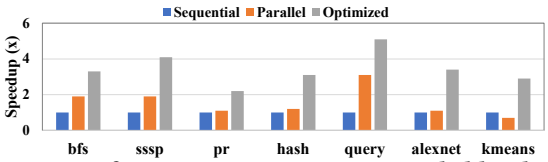


Figure 10: Performance improvement provided by the proposed optimization.

all computations in this sub-graph; otherwise, we adopt parallel aligned method to allocate memory. Cost models for sequential and parallel methods for a sub-graph are: $Cost_{seq} = Cost_{op} * N_{op}$ and $Cost_{par} = Cost_{op} * N_{par\_step} + Cost_{mv} * N_{mv}$ where $Cost_{op}$ and $Cost_{mv}$ denotes the latency of one vector operation and movement, N_{op} denotes the total number of operations, N_{par_step} denotes the number of steps required for maximum parallelism, and N_{mv} denotes the number of data movements required for maximum parallelism. All these values can be estimated by the runtime information (e.g., data size and operation type) and hardware parameters.

We compare the proposed data allocation algorithm to the sequential aligned allocation and the parallel aligned allocation for all tested applications in the single-chip system with 32 channels and each channel has 16 memory banks, as shown in Figure 10. Compared to the parallel aligned allocation, the proposed method improves performance by 2.2 ×. Such results come from the fact that some computations are bounded by data movements when using parallel aligned allocation. In this case, the propose method selects the sequential aligned method which reduces the data movement overhead at the cost of less parallelism.

4 CONCLUSION

In this work, we propose a full-stack simulation infrastructure, DP-Sim, to evaluate different design choices in DPIM architectures from both software and hardware. We design the whole stack of simulating DPIM systems, including a software library, a configurable compiler layer, and a fast but accurate PIM-enabled architecture model. Our experiments show that DP-Sim provides 10.3× faster simulation with only 6.3% result difference as compared to a validated memory simulator. Furthermore, we utilize DP-Sim to explore various DPIM-specific design choices and our experiments show that such design space has a significant impact on the performance of DPIM-based acceleration.

ACKNOWLEDGMENT

This work was partially supported by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and also NSF grants #1730158 and #1527034.

REFERENCES

[1] Shaizeen Aga et al. 2017. Compute caches. In *HPCA'17*. IEEE, 481–492.
[2] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).
[3] Trevor E. Carlson et al. 2014. An Evaluation of High-Level Mechanistic Core Models. *TACO'14*, Article 5 (2014), 23 pages. https://doi.org/10.1145/2629677
[4] Charles Eckert et al. 2018. Neural Cache: Bit-serial In-cache Acceleration of Deep Neural Networks. In *ISCA'18*. IEEE Press, Piscataway, NJ, USA, 383–396.
[5] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-Memory Data Parallel Processor. In *ASPLOS'18*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3173162.3173171
[6] Fei Gao et al. 2019. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. In *MICRO'19*. ACM, New York, NY, USA, 100–113. https://doi.org/10.1145/3352460.3358260
[7] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 751–764. https://doi.org/10.1145/3093337.3037702
[8] Mohsen Imani et al. 2019. FloatPIM: In-memory Acceleration of Deep Neural Network Training with High Precision. In *ISCA'19*. ACM, New York, NY, USA, 802–815. https://doi.org/10.1145/3307650.3322237
[9] Byung Chul Jang et al. 2018. Memristive Logic-in-Memory Integrated Circuits for Energy-Efficient Flexible Electronics. *Advanced Functional Materials* 28, 2 (2018), 1704725.
[10] Y. Kim, W. Yang, and O. Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (Jan 2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[11] Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342* (2018).
[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
[13] Shahar Kvatinsky et al. 2015. VTEAM: A general model for voltage-controlled memristors. *IEEE Transactions on Circuits and Systems II: Express Briefs* (2015).
[14] Sheng Li et al. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172
[15] S. Li et al. 2017. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 288–301.
[16] Chi-Keung Luk et al. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.* 40, 6 (June 2005), 190–200. https://doi.org/10.1145/1064978.1065034
[17] Micron. [n.d.]. DDR4 SDRAM System-Power Calculator. https://www.micron.com/~/media/documents/products/power-calculator/ddr4_power_calc.xlsm?la=en.
[18] Naveen Muralimanohar et al. 2007. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *IEEE/ACM Micro*.
[19] Matthew andothers Poremba. 2015. Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems. *IEEE Computer Architecture Letters* (2015).
[20] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10, 1 (Jan 2011), 16–19. https://doi.org/10.1109/L-CA.2011.4
[21] A. Shafiee et al. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 14–26.
[22] Linghao Song et al. 2018. GraphR: Accelerating graph processing using ReRAM. In *HPCA'18*. IEEE, 531–543.
[23] J. Wang et al. 2020. A 28-nm Compute SRAM With Bit-Serial Logic/Arithmetic Operations for Programmable In-Memory Vector Computing. *IEEE Journal of Solid-State Circuits* (2020).