Programming Models for Efficient Data Orchestration in Memory-Centric Systems

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Eric Tang

B.S., Electrical and Computer Engineering, Cornell University

Carnegie Mellon University

Pittsburgh, PA

September 2026

© Eric Tang, 2026

All Rights Reserved

Acknowledgments

This thesis would not have been possible without the support of an incredible number of people. Firstly, I would like to thank my advisors, Franz Franchetti and James Hoe, for their incredible guidance and support throughout my PhD journey. Franz and James, you not only taught me how to ask insightful questions and how to do research but also how to be kind and supportive mentors. While I came into the program with a different idea of what I wanted to research, I was given the opportunity to explore across the computing stack and for that I learned and grew a lot more. I would also like to thank to my thesis committee members and collaborators, Prof. Kevin Skadron and Prof. Phil Gibbons for their valuable questions, insights, and feedback which helped to improve this research and the final dissertation.

I have had the privilege of working with many brilliant collaborators who helped to make the work in this thesis more compelling. When working with PIM systems, Yiwei Zhao was instrumental in getting set up and optimizing algorithms on UPMEM hardware. Farzana Siddique, William Bradford and Deyuan Guo from UVA all provided important insights and discussions when co-designing for PIM and building a PIM simulator to perform our analysis. Furthermore, during my earlier work on Magic Memory, Gustav Hansen and Christopher Stange were invaluable in figuring out low-level system details to make the prototype a success.

The members of my research groups were essential for making my time at CMU both academically successful and a fun place to be. I would like to thank

the SPIRAL group members including Het Mankad, Guanglin Xu, Sanil Rao, Peter Oostema, Naifeng Zhang, Emma Quinn, Quentin Oschatz and Tianyun Zhang as well as those in the James Hoe group including Shashank Obla, Siddharth Sahay, Chengyue Wang, Jason Cheung, Will Krska, Simon Yu, Edwin Lim, and Sanjay Seshan. From our research discussions and practice talks in the office to exploring different ways of brewing coffee and everything else that campus had to offer, you made this journey possible. While we all met as fellow students at CMU, many of us have become friends outside of school as well which made A-level a place I looked forward to going to. I would also like to thank Prof. Tze Meng Low for his many provocative and insightful conversations around the office which really helped to shape my outlook on high performance computing.

As I began my PhD during COVID, it is abundantly clear that this journey is far more enjoyable with friends. I want to acknowledge those I met around A-level and CMU including Upasana Sridhar, Elliott Binder, Icey Chen, Siyuan Chen, Pavan Hegde, and Chris and Jen Talbot. Whether it be playing basketball, hiking and biking, exploring all that Pittsburgh has to offer, or just hanging out at someones' apartment you all made Pittsburgh a fun place to live. I am equally grateful for all my friends Srikar, Brett, Siddhant, and Gaurang who came out to Pittsburgh and Mahiro, Junlan and others for your support during this time.

Finally, I would like to thank my family. To my mom and dad, thank you for your continued support and encouragement throughout my PhD. My brother, Larry, who embarked on this journey alongside me and was surprisingly supportive during this process. And my girlfriend, Lauren, who always supported me throughout the highs and lows over the years. From many late night calls to frequent trips back and forth to Pittsburgh, I could not have done this without you.

This work was supported over the years in part by PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program spon-

sored by DARPA, the U.S. Department of Energy under award number DE-AC05-00OR22725, the Intel/VMware Crossroads 3D-FPGA Academic Research Center and by the CIT Dean's Fellowship.

Eric Tang

Carnegie Mellon University

September 2026

Abstract

Memory-centric systems, such as Processing-In-Memory (PIM) architectures, place computation where data are stored. This suggests that for applications such as graph analytics and sparse linear algebra which are bottlenecked by data movement, these systems could provide significant improvements. However, realizing these gains is challenging. Memory-centric systems span a wide design space, from PIM devices with restricted arithmetic capabilities to accelerators whose memory is shared with the host, and programming these devices requires managing the data placement and movement on the machine. These computational constraints make it difficult to identify which operations or algorithms can be effectively mapped to this hardware. This work shows that a programming model using a declarative specification over regions of memory carries enough information for memory-centric systems to efficiently manage data orchestration and enough capability for data-movement bound iterative algorithms to be reformulated within this model.

Hardware acceleration approaches placed near or in memory can be missing basic arithmetic capabilities such as multiplication or floating point support due to tight area and power constraints. However, the iterative optimization algorithms that are bottlenecked by data movement require these basic operations in order to converge towards a solution. Filter and update operations are simple enough to execute efficiently on hardware with limited arithmetic and can reduce the amount of data movement that is required in these iterative algorithms. For iterative al-

gorithms, the filter produces a sparse output while the update consumes a sparse input which enables the traffic across the memory bus to scale with the number of elements selected rather than the size of the dataset. By using a memory-side filter operation, each iteration computes the update using only the elements that most affect the residual. The filter update based programming model allows for these operations to be expressed over regions of memory, which then enables the system to partition the data across near memory computation, reduce data movement across the memory bus and lower these operations to the specific hardware target. To demonstrate the effectiveness of this approach, existing algorithms like gradient descent and new filter-update modified algorithms like Gauss-Seidel are expressed using this programming model and implemented on a real PIM system. The filter-update programming model reduces total data movement and shows up to 6.3x speedup when compared to the standard algorithm.

For systems where more capable logic shares an address space with the host, the Magic Memory programming model enables programmers to describe computation by declaring invariants between memory regions. In this model, the programmer can access these memory regions using ordinary loads and stores, and the system remains responsible for the physical representation of the data, the computation to be performed to maintain the invariant, and the data movement between devices. This is demonstrated using interactive PageRank where a sparse graph is maintained behind a dense interface and the rank vector is recomputed as necessary when edges are added or removed. The specification of the computation between regions of memory allows for the Magic Memory program to abstract the sparse format used for the graph and how and when the computation occurs.

Contents

Acknowledgments **iii**

Abstract **vi**

List of Figures **xii**

List of Tables **xvii**

Chapter 1 Introduction **1**

1.1 Motivation 2

1.2 Background 4

1.3 Approach 7

1.3.1 Filter-Update Model 7

1.3.2 Magic Memory 9

1.4 Contributions 12

1.5 Thesis Outline 13

Chapter 2 Background **15**

2.1 Types of Processing-In-Memory Architectures 15

2.1.1 Processing-Using-Memory (PUM) 16

2.1.2 Processing-Near-Memory (PNM) 18
2.2 Programming Models for PIM Architectures 21
2.2.1 Accelerator Programming Model 21
2.2.2 Library Model 25
2.2.3 Summary 26
2.3 Sparse Linear Solvers 27
2.3.1 Jacobi Iteration Method 29
2.3.2 Gauss-Seidel 30
2.3.3 Successive Over-Relaxation 31
2.4 Summary 32

Chapter 3 Filter Update Programming Model **33**
3.1 Filter-Update Overview 33
3.2 Filter Update Programming Model Compute Stack 36
3.2.1 Memory Query Language (MQL) 38
3.2.2 MQL Hardware Abstraction Layer 41
3.2.3 System Software and Runtime 46
3.2.4 Filter Update Operations on PIM Architectures without Floating Point Support 47
3.3 Target Hardware 48
3.3.1 UPMEM 48
3.4 Summary 55

Chapter 4 Filter-Update Applications **57**
4.1 Filter-Update Expressible Workloads 57
4.2 Gradient Descent 60

4.2.1 Data Movement Reduction Analysis 61
4.2.2 Filtering Approaches 65
4.2.3 Potential Performance Gain 66
4.3 Filter-Update Modified Linear System Solvers 68
4.3.1 Types of Filtering Approaches 69
4.3.2 Coloring the Matrix 71
4.3.3 Relaxation Constant 73
4.3.4 Summary 73
4.4 Filter Update Solver Performance Evaluation 74
4.4.1 Speed-of-Light Performance Model 76
4.4.2 Peak Filter-Update Performance Evaluation 76
4.5 Summary 85

Chapter 5 Magic Memory **87**
5.1 Magic Memory Programming Model 88
5.2 Magic Memory API 92
5.3 Magic Memory Consistency 95
5.4 Sparse Data Structures 96
5.4.1 Dense Interface for Sparse Data 97
5.4.2 Physical Representation and Format Selection 98
5.4.3 Managing the CSR Backend 98
5.5 Graph Degree 100
5.5.1 Differential Maintenance 101
5.6 Prototype Magic Memory System 102
5.7 Magic Memory Example: PageRank 105
5.8 Evaluation 107

5.9 Summary 110

Chapter 6 Conclusion **111**

6.1 Lessons Learned 114

6.2 Future Work 116

6.2.1 Designing New PIM Hardware 116

Bibliography **119**

List of Figures

1.1 The high latency and limited bandwidth connection between the host processor and main memory is the main bottleneck in applications ranging from machine learning to graph algorithms to scientific computing. 3

1.2 Magic Memory programming model for tracking the triangle count of a graph. One memory region is described as storing the graph while the other stores the number of triangles in the graph described by the input region. As stores to the graph change the structure, the number of triangles in the output magic memory region is automatically updated. 10

2.1 UPMEM PIM System. UPMEM PIM-enabled memory resides in a separate DDR4 channel from the host memory. Each PIM chip has 8 DRAM Processing Units (DPUs). Within each DPU is a RISC core that has both instruction RAM and working RAM (for staging data) and a DMA engine for copying data to and from the main DRAM bank. 20

2.2 The accelerator model in outline. The host partitions, pads, transfers, launches, gathers, and reduces; the device kernel stages tiles between its local DRAM and scratchpad by hand. Every data-movement decision in both listings is written by the programmer. 23

3.1 Illustration of data movement in the filter-update programming model. PIM-enabled memory is used to evaluate basic filter and update operations while the traditional host CPU is used to perform more complex arithmetic operations which do not work well in memory like multiplication and floating point arithmetic. 35

3.2 PIM Compute Stack 37

3.3 Equivalent SQL query for a filter operation. 39

3.4 Equivalent SQL query for an update operation 40

3.5 UPMEM filter operation throughput for various filtering percentages across a range of vector sizes 50

3.6 UPMEM filter operation throughput while only accounting for DPU to host DRAM communication time. 52

3.7 UPMEM update operation throughput for various filtering percentages across a range of vector sizes 53

3.8 UPMEM update operation throughput while only accounting for DPU to host DRAM communication time. 54

4.1 Cumulative data movement until convergence over iterations for different gradient descent methods with varying sparsification strategies. The points on each line mark the iteration where the residual falls below 10^{-j} for j from 3 to 7. The threshold-based filtering approach requires least data movement for each isoline with as much as 3.9x less when compared to full gradient descent. The random filtering approach requires about the same amount of data movement as Full GD but 10x more iterations. 61

4.2 Convergence behavior of gradient descent under different gradient sparsification strategies on an ill-conditioned quadratic objective function. The residual computes the error between the value of x_t in iteration t and the known minima x_*. x_t is considered to have converged when the residual $\leq 10^{-7}$. The filter approaches both converge at a rate comparable to the full gradient descent while requiring less data movement. 62

4.3 Runtime for solving a linear system using Gauss-Seidel versus a speed of light model for Filter-Update modified Gauss-Seidel. Both algorithms are run until convergence. 77

4.4 Runtime for 5,000 iterations of Gauss-Seidel vs. Filter-Update modified Gauss-Seidel across synthetic matrices of various sizes. The runtime when using a threshold-based filter that is set such that about 1% of the dimensions are used to compute the update in each iteration is as much as 24.1x faster based on a speed-of-light performance model for the filter and update operations. 81

4.5 Runtime for 1,000 iterations of Gauss-Seidel vs. Filter-Update modified Gauss-Seidel across synthetic matrices of various sizes. The runtime when using a threshold-based filter that is set such that about 1% of the dimensions are used to compute the update in each iteration is as much as 40.2x faster based on a speed-of-light performance model for the filter and update operations. 82

4.6 Runtime for Gauss-Seidel and Filter-Update modified Gauss-Seidel while using the $\|\mathbf{r}^{(k)}\|_2/\|\mathbf{b}\|_2 < 0.5$ stopping condition. The threshold is set such that approximately 1% of the data is passed through the filter in each iteration. Under this earlier stopping condition, the filter-update approach achieves a speedup of up to 7×, slightly exceeding the performance gains observed under traditional convergence criterion. 84

5.1 For x86 systems, Magic Memory regions utilize the virtual memory addresses outside of the user space and the kernel space. Here, the input Magic Memory region is a dense matrix which represents a graph while the output Magic Memory region is a vector with the PageRank of each node of the graph. 90

5.2 Dense versus sparse view of a matrix as seen from Magic Memory. Logically from the programmer's perspective, the entire matrix is available and stored in memory but Magic Memory hides the compressed physical representation that only tracks the nonzero elements. 97

5.3 Magic Memory on CPU-FPGA System. 103

5.4 Comparison of SpMV kernel on FPGA versus baseline CPU implementation. All graphs used to generate the sparse matrix have 100,000 nodes which makes the SpMV operation a 100k x 100k sparse matrix multiplied by a 100k dense vector. By varying the number edges in graph, the sparsity of the matrix is changed 108

6.1 Potential bank-level PIM hardware for supporting Filter-Update operations. In this microarchitecture the SIMD width is matched to the size of the global row buffer and global data line (GDL) in order to support peak bandwidth. 117

List of Tables

3.1 Filter predicates provided by the hardware abstraction layer. 41

3.2 Update operators provided by the hardware abstraction layer. 44

3.3 Operations provided at the hardware abstraction layer. Bracketed arguments are optional. 45

4.1 Bandwidth reduction factors for different sparsity levels 64

4.2 Convergence and Data Movement for Different Filtering Techniques 65

4.3 Iteration counts to convergence for Filter-Update Gauss-Seidel, comparing threshold-based and top-k filter selection across scaled `bcsstk08` and `bcsstk09` systems. 71

4.4 Baseline Gauss-Seidel versus the Filter-Update threshold solver on replicated `bcsstk08` [1] systems. Runtimes are wall-clock seconds to a relative residual below 10^{-6}. 78

4.5 Peak machine DRAM performance 78

5.1 The Magic Memory API. Allocation and initialization mirror the C standard library; the remaining functions declare invariants between regions and configure their representation. 94

5.2 Region properties set through `mx_set_property`. 95

5.3 Operations on the Magic Memory dense interface and their effect on the underlying CSR representation. nnz_i denotes the number of stored nonzeros in row i. 99
5.4 Memory Mapped FIFO Latency 104

Chapter 1

Introduction

The traditional von Neumann architecture separates computation from memory which has become a fundamental bottleneck in modern applications as the two technologies scaled at different rates and data requirements for many algorithms soared [2–4]. To alleviate this bottleneck, memory-centric systems, such as Processing-In-Memory (PIM) architectures, have emerged to perform computation closer to where the data are stored [5]. However, realizing this performance remains difficult given the constraints of near memory computation [6] and the difficulty in programming these architectures [7–9].

The thesis of this dissertation is that a programming model where computation is expressed as a declarative specification over regions of memory is sufficient for data-movement bound iterative algorithms to exploit memory-centric systems. On the algorithm side, exposing a specification of the operation on regions of memory provides a high level abstraction for the computation that is to be targeted. This makes it simple for domain experts to use these operations to utilize near memory hardware acceleration. On the system side, a declarative specification allows the

system to know which data to compute upon and the relationship between elements rather than requiring directives on how to move the data. Since the computation is described over a region of memory, the system is able to partition, place and transfer the data appropriately between devices without lower level instructions from the program. The models presented here are thus shaped by both what the hardware can execute efficiently and the algorithms are reshaped by what the model can express.

1.1 Motivation

As modern applications have scaled in size, algorithms across a wide range of domains such as machine learning [10, 11], graph analysis [12], and scientific computation [13] have become bottlenecked by data movement requirements [14]. To this end, a myriad of memory-centric systems [15] have been developed that place compute directly in or near memory to eliminate or reduce the data movement requirements between the host and memory. These systems span a wide design space from Processing-In-Memory approaches where compute is performed in or near main memory to systems where more capable compute or dedicated accelerators are presented to the host through a shared address space that can be accessed via loads and stores [16–20].

With this diversity of hardware, memory-centric systems remain difficult to program. This is due to the fact that these systems vary in terms of compute capability and how the memory is exposed to the host [21]. With respect to compute capability, approaches differ in terms of where compute units are placed, how many there are, and which arithmetic operations are supported. As a result, these devices vary in terms of the interface that is exposed to the host device and application

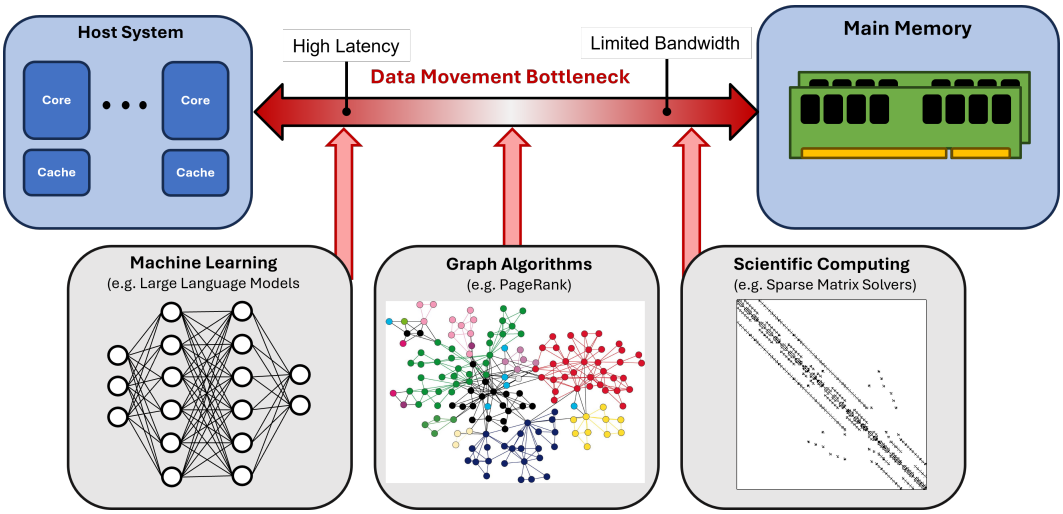

Figure 1.1: The high latency and limited bandwidth connection between the host processor and main memory is the main bottleneck in applications ranging from machine learning to graph algorithms to scientific computing.

developer. For the interface presented to the host processor, near-bank PIM devices expose a partitioned, non-coherent address space in which a compute unit may access only its local memory [22, 23]. In DIMM-based general-purpose PIM, all host communication occurs through explicit bulk transfers, and inter-unit communication must be routed through the host [24, 25]. Devices that integrate compute at the bank level within a standard DRAM interface instead accept PIM commands through ordinary memory operations, but retain the same bank-local restriction on what each compute unit may address [26]. Other devices, like those attached via CXL [27, 28], share an address space with the host and can communicate through ordinary loads and stores. However, both classes frequently require vendor-specific libraries. With these differences in hardware and software approaches, application developers are forced to familiarize themselves with the microarchitectural details of the hardware being targeted in order to implement an algorithm efficiently.

A higher-level programming model is necessary to effectively utilize memory-

centric hardware [7, 29]. Without one, each application would require a hardware expert to explicitly manage data movement and describe the computation for every target. Higher-level abstractions instead allow programmers to express computation in a manner that is independent of any specific architecture, using operations expressive enough for the system to lower to a particular target.

The appropriate abstraction depends upon the capabilities of the hardware in the system. When mapping an algorithm to a memory-centric system, the constraints and performance of the hardware inform not just where computation could be performed but rather where it should be performed. For many PIM devices, arithmetic operations such as multiplication or division are very expensive or floating-point operations are not supported [30, 31]. In addition, computation must occur between elements that are aligned and shifting elements around could even require a round trip to host memory. In these cases, while it is theoretically possible to perform these expensive operations or expensive data manipulation on the PIM device, a good programming model for a memory-centric system would not enable this.

The interface between the host and device is another important constraint that determines what the programming model exposes. When the memory-centric computation occurs on a device where the memory is shared with the host, the physical data movement can be hidden from the programmer. Instead, the programmer can focus upon a logical view of the system while the physical mapping of the computation is determined by the system.

1.2 Background

Memory-centric systems have been proposed and, more recently, built across a wide range of the memory hierarchy. At one end, processing-using-memory designs per-

form computation within the DRAM array itself, exploiting the charge-sharing behavior of the bitlines to implement bulk bitwise operations [30, 32]. Nearer the DRAM interface, processing-near-memory designs add discrete logic. This includes fixed-function multiply-accumulate units adjacent to each bank in commercial high-bandwidth parts [33, 34], general-purpose in-order cores at the bank level in UPMEM [35], and full processor cores in the logic layer of stacked memories [36]. A separate line of work places compute behind a shared address space rather than inside the memory device, allowing an accelerator and its local memory to be accessed by the host through ordinary loads and stores [37]. These designs differ in how much computation they can perform, how many units are available, and whether those units can communicate, and Chapter 2 examines these differences in detail.

Programming this hardware is typically done in one of a few ways. The first is to write low-level code against a vendor SDK. On UPMEM, the programmer writes a kernel for the DRAM processing units, partitions the input across the units and copies each portion into the unit's local DRAM bank, issues an explicit blocking DMA transfer from that bank into a scratchpad before the data can be operated upon, launches the kernel, and copies partial results back to the host to be combined, since the units cannot communicate with one another [38]. Every one of these steps exposes a property of the machine. The transfer between the local bank and the scratchpad is managed entirely by the programmer, who must choose a tiling of the data that fits within the scratchpad's capacity and must overlap transfers with computation by hand if the pipeline is to remain occupied. The kernel itself is bounded by the capacity of the instruction memory, which is small enough that the complexity of a computation becomes a constraint on what can be offloaded. The absence of communication between units means that any algorithm requiring

a reduction or an exchange of partial results must be restructured around a host round trip. A program written this way encodes the number of units, the scratchpad capacity, the instruction memory limit, and the transfer granularity of one device, and none of these assumptions survive a change of target.

Processing-using-memory approaches impose a different and more restrictive requirement. Because an in-array operation combines the rows of a subarray as the sense amplifiers activate them, its operands must already be physically resident within the same subarray, and aligned within it, before the operation can be issued [32]. Bit-serial designs additionally require that the bits of each operand be laid out vertically across rows rather than horizontally along one, so data must be transposed on entry and transposed back before the host can interpret it [30]. Physical data placement is therefore the responsibility of the programmer and requires expressing the operations in terms of subarrays, rows, reserved operand rows. The cost of arranging data to meet these conditions is charged against every operation rather than against the dependences of the algorithm, so an algorithm whose operands are not naturally co-resident may find that relocation dominates its execution entirely.

A third approach hides the hardware behind an existing library interface. For Samsung's HBM-PIM a runtime analyzes an application written against a conventional framework such as PyTorch or TensorFlow, identifies the operations it can accelerate, maps their operands into the PIM address space, and generates and invokes the corresponding microkernels [33]. While the programmer is able to write against ordinary framework code this approach is unable to perform any operations the runtime does not already recognize.

Each of these approaches trades off between the expressiveness of the oper-

ations and abstraction. Writing against a vendor SDK allows for more arbitrary computation but requires the programmer to be familiar with vendor-specific tools and to manage data movement explicitly, both between the host and PIM-enabled memory and among the units themselves. PUM approaches go further and make proper physical placement essential not only for performance but also correctness. The library model removes these concerns entirely, but only for the operations that have already been implemented. To enable both leveraging memory-centric hardware acceleration and high level abstractions the programming model requires a declarative specification that is abstract enough for the system to understand and expressive enough to describe the computation the programmer intends. These specifications must be determined by what the hardware can execute efficiently and afterwards the algorithm must be reformulated to fit within those constraints. The programming model then provides enough information for the system to determine the placement and movement of data. The remainder of this chapter introduces two programming models that make this exchange, and the algorithms that were reformulated to fit them.

1.3 Approach

1.3.1 Filter-Update Model

For systems where in-memory compute capability is limited, the Filter-Update model expresses computation as a declarative specification over a region of memory using two operations. A filter selects the elements that will contribute to the current iteration while an update accumulates values to a memory region. This specification states what data are selected and what data are accumulated but does not describe how either operation is performed in hardware. For iterative algorithms, the compu-

tation in each iteration can be adapted such that the filter operation selects a subset of the data that is transferred across the memory bus, the host processor uses this data to compute a sparse update and the update operation manages transferring this data from the host back to memory where it is accumulated with the original data.

These operations remain basic enough that they can be mapped to even more constrained PIM hardware. Each of the filter and update operations require only basic boolean logic or a single accumulation operation. Neither operation requires a processing unit to retain state beyond the element it is currently examining, which keeps the model within the arithmetic and instruction-capacity limits of devices such as UPMEM and removes any need for units to communicate with one another. As the filter produces a sparse output while an update consumes a sparse input, the data crossing the memory bus is proportional to the elements selected rather than to the size of the working set. The number of processing units, the partitioning of the region across them, the transfers between each unit's local memory and its scratchpad, and the point at which results return to the host are all decided by the system rather than stated by the programmer. How these hardware parameters can be inferred from these operations is discussed in Chapter 3 where the compute stack and API for this programming model is introduced and outlined.

This constraint also informs the algorithm design as well as the hardware. Algorithms must be reformulated so that each iteration's update is computable from a filtered subset of the working set. Prior work has established that many such methods tolerate operating on a subset of their working set each iteration, converging in more iterations while performing less work in each [39, 40]. This dissertation restructures several such methods so that the subset is chosen by a

filter and the update is computable within the partitioning the model imposes, and characterizes the cost in iterations that the restructuring incurs. While not every method is able to be recast into this model, Chapter 4 highlights not only the algorithms which work well with the filter update model but also identifies the properties that determine which do not.

1.3.2 Magic Memory

Where more capable logic is available, and where the memory attached to it is exposed to the host through a shared address space, the abstraction can be more generalized. The Magic Memory programming model allows the programmer to declare an invariant between regions of memory and allows these regions to be accessed with ordinary loads and stores as illustrated in Figure 1.2. The system is then able to automatically maintain that invariant. Because the invariant relates regions rather than naming operations, the system observes the host's accesses and determines from them what work is required and when. This then enables operations to be performed on dedicated hardware accelerators present in the system and allows for the flexibility to perform the required computation eagerly or lazily. It may recompute eagerly as soon as the source region is modified, or defer the work until a load from the derived region requires it. The programmer states no schedule, issues no kernel launch, and marks no synchronization point.

The system further owns the physical representation of the data. Each magic memory region logically represents an N-dimensional tensor. This allows for algorithms to be described using high-level operations rather than needing to be adapted for various different physical representations of the data. For graph algorithms and other sparse data, this is often especially tedious given the wide range of different

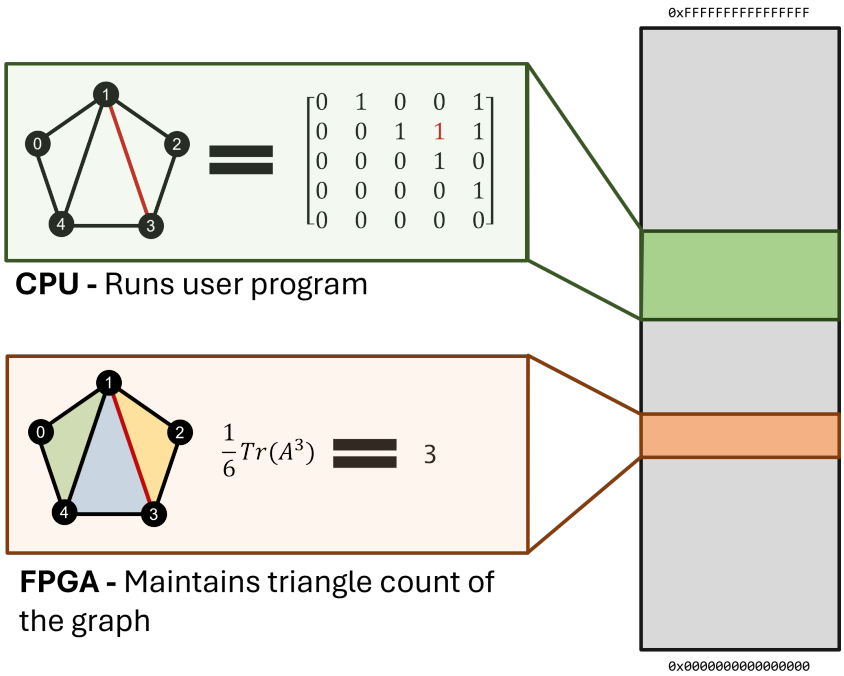

Figure 1.2: Magic Memory programming model for tracking the triangle count of a graph. One memory region is described as storing the graph while the other stores the number of triangles in the graph described by the input region. As stores to the graph change the structure, the number of triangles in the output magic memory region is automatically updated.

data formats that are frequently used depending upon the structure of the sparsity and the computation to be performed. While in dense arrays, the physical location of an element simply depends upon the indices, looking up an element in a sparse data structure often requires a sequence of data dependent load transactions as the number and location of other nonzero elements affect the placement of each element. Furthermore, insertion into sparse data structures often requires not only displacing the elements surrounding but also updating any and all auxiliary data structures used to track where the nonzero elements reside. As various arithmetic operations are performed using these sparse data structures, different sparse formats are used depending upon the access pattern that is required by the computation and by certain hardware parameters such as the memory hierarchy, cache size and type of compute units that are available. As the Magic Memory programming model describes both the data that lives within a memory region and the computation to be performed, this complexity can be hidden from the programmers and removes the need to understand the hardware in order to get good performance and pushes the responsibility to the runtime.

The model suits applications in which a derived quantity is read far more often than its inputs change, and in which the cost of maintaining it incrementally is small relative to recomputing it. Interactive graph analytics is the case developed in this dissertation. Chapter 5 describes an interactive PageRank in which the graph structure is hidden from the host behind a dense interface and the rank vector is recomputed automatically as edges are added and removed.

1.4 Contributions

The thesis statement of this dissertation is: In memory-centric systems, describing computation through declarative specifications between memory regions enables efficient hardware mapping while preserving the flexibility required to develop novel algorithms.

To substantiate this claim, this work demonstrates how hardware-software co-design of these memory operations unlocks the acceleration potential of processing-in-memory architectures. By bridging application-level requirements with underlying hardware constraints, this dissertation makes the following contributions:

- Identifies filter and update operations as key operations that can not only be mapped to a wide range of PIM architectures but also obtain meaningful speedup by being performed in memory.
- Demonstrates that filter and update operations can be used effectively in common iterative optimization algorithms
- Characterizes the trade-off between data movement and iteration count when using a fraction of the dataset in each iteration
- Establishes that the maintenance of a data structure's physical representation can be relocated from the program to the memory system, and identifies the properties a system must possess for this to be possible: that device memory be addressable by the host, and that the memory-side logic be general enough to maintain a structure rather than to apply a fixed operation.
- Shows that a declarative specification carries sufficient information for a system to determine what data must move and when, without the program sup-

plying a schedule. By tracking individual loads and stores to memory, computation can be performed eagerly or lazily to maintain the appropriate state.

1.5 Thesis Outline

Chapter 2 provides a background on existing processing-in-memory systems and describes how computation is typically performed on this hardware.

Chapter 3 describes the design of the filter update programming model for memory-centric systems where there is limited compute capability near memory. This chapter outlines how the specification of filter and update operations can be mapped to a high level API and what the compute stack looks like for lowering this specification to hardware. The details for how these operations can efficiently map to both current and future PIM hardware is discussed here.

Chapter 4 illustrates how the filter update programming model informs the algorithm design and the various algorithms that were adapted using the filter update programming model. This is evaluated using two such iterative algorithms including gradient descent and a filter-update modified Gauss-Seidel algorithm. Here both the performance and effect on convergence are evaluated.

In Chapter 5 the Magic Memory programming model is presented for memory-centric systems where more general purpose computation is available. This chapter views memory using a spreadsheet-like view and uses a declarative specifications between regions of memory can be used to describe computation. The utility of this approach is demonstrated with the abstractions on sparse data structures and eager versus lazy computation that this enables. A prototype of the Magic Memory programming model for an interactive PageRank application on a CPU-FPGA system is outlined.

The last chapter discusses the lessons learned from this dissertation and potential future directions for continuing this work.

Chapter 2

Background

In this dissertation, a hardware-software co-design approach is taken in order to determine the appropriate specifications to declare upon regions of memory. For this reason, in order to design for memory-centric systems, a background on existing near memory hardware like the class of PIM architectures ranging from Processing-Using-Memory to Processing-Near-Memory is provided first. Then, existing programming models for targeting these architectures is covered and compared with the approrach presented here. Following this, a basic description of common iterative optimization algorithms that will be used later is provided as well. These basic iterative optimization algorithms are used as a starting point for when the programming models presented in this dissertation are used to target the same problem but leverage PIM hardware acceleration to alleviate common bottlenecks.

2.1 Types of Processing-In-Memory Architectures

PIM architectures refer to those that integrate computation directly where data is stored. This reduction in data movement can lead to an improvement in both

performance and energy consumption. However, the placement of compute units within the DRAM hierarchy can vary dramatically, with each location imposing a unique set of constraints. These variations have made it difficult to target these systems with a single unified programming method. Depending upon where and how the compute is placed near memory, PIM architectures are commonly classified into Processing-Using-Memory (PUM) and Processing-Near-Memory (PNM) [41].

2.1.1 Processing-Using-Memory (PUM)

PUM architectures describe PIM architectures where the memory subarray and peripheral circuitry perform the computation. In DRAM this could be done using charge sharing on the bitline as seen in (Ambit [32], SIMDRAM [30], etc...). By performing computation within the DRAM cell array, these architectures can take advantage of the higher internal bandwidth that is available. In designs such as Ambit [42], they have been shown to enable bulk bitwise operations in DRAM. This is done by performing a triple-row activation where it lends itself to a majority of three operation and naturally leads itself to SIMD computation. While this has been shown to significantly improve both energy efficiency and operation throughput, programming this architecture requires both breaking down the computation into very wide SIMD bitwise operations and ensuring the operands are physically placed within the same subarray in DRAM.

In SRAM bitline-based computation can also be used to enable processing using memory. In this, multiple word lines can be asserted at the same time to implement an AND or NOR operation [43]. Compute caches exploit this logic at the L3 layer to enable bitwise boolean logic. This logic was later expanded to support general purpose compute by in the neural cache [44] or the duality cache [45].

Relative to DRAM, SRAM offers non-destructive reads and lower latency read and write operations. At the cache level, these operands are also already connected within the existing coherence protocols. However, the same constraints that affect DRAM still apply here in terms of needing to control the physical data placement and the fixed data width of the operands and operations. On top of this, the most restrictive aspect of this approach remains that the data must be able to fit within the cache which prevents these approaches from scaling to problem sizes that exceed the cache capacity.

In ReRAM or PCM devices, analog computation can be added near memory to enable analog MVM (matrix vector multiplication) using Kirchhoff's and Ohm's laws. When vector elements are supplied to the word lines as supply voltages and the matrix values are mapped to the ReRAM crossbar, the resulting supply voltages hold the result of a vector matrix operation as determined by Kirchkoff's law [46, 47].

These approaches have a number of constraints that make them difficult to adopt in a hardware agnostic fashion. For those that work by using multiple DRAM rows, every operation is destructive. This means that operands must perform a row clone before performing the operation. None of these approaches are able to support general purpose arithmetic operations. This means that operations such as multiplication must be emulated using bit-serial steps and sequences of operations. For bit-serial approaches, the data must be transposed before writing them to memory. In addition, these approaches require the data and operands to be physically aligned within memory. This then forces programmers to be aware and control the physical memory placement. In addition, the SIMD width of these operations is fixed by the subarray size. Any operations that do not fully make use of this lead to wasted operations. These approaches are unable to support any control flow and

cross column data movement is not possible. For this reason, reductions and gather scatter operations are not possible.

2.1.2 Processing-Near-Memory (PNM)

PNM architectures add additional compute logic near the DRAM interface. Depending upon where this logic is placed, the parallelism that is available and the compute that can be performed varies. For logic placed closer to the DRAM subarray, more parallelism and higher bandwidth is available at the cost of less capable compute and less data that is accessible [21]. As compute is moved further away from the storage technology, these constraints are lessened. This tradeoff between compute capability and data storage can be seen across the following common PNM approaches.

Subarray-level bit-serial architectures perform compute at the subarray level but introduce new hardware near the row buffer that performs basic arithmetic operations on individual bits of data at a time. The data is stored vertically within a subarray to enable subarray row level parallelism and thus requires a sequence of bitwise operations in order to perform arithmetic operations for integer or floating-point data types [48]. This imposes constraints that require programmers to manage physical data placement and layout, while also introducing new considerations regarding the cost of performing complex operations such as integer multiplication or floating point arithmetic. For other non bit-serial subarray-level PNM architectures [49] where the data layout is horizontal across the row buffer more traditional SIMD approaches can be used for programming. However, in these cases, the data width is important in determining the number of elements that can be processed.

For processing units placed at the bank interface, more general purpose com-

putation can be achieved via bank level parallelism [25, 35]. This type of approach is used by commercial systems such as UPMEM [35] where a simple in-order RISC core is placed next to the bank. While more general purpose programs can be written for these processing units, the user must still explicitly manage data movement between the host CPU and the PIM-enabled memory along with managing the kernel launch on the bank-level compute units. Furthermore these approaches remain restricted in that the compute units next to each bank are unable to communicate with one another and require utilizing the host CPU to merge partial results or manage communication between banks.

At the DIMM and rank level, compute logic can be integrated into the buffering circuitry of conventional memory modules without modifying the DRAM arrays themselves [50, 51]. PNM systems also include 3D-stacked architectures, which add a logic layer to layers of vertically integrated DRAM dies [36]. As the compute logic moves further from the DRAM array, substantially more capable programmable or reconfigurable compute units can be provided, making these architectures more programmable using using traditional multicore approaches.

UPMEM. UPMEM [24] is a commercially available Processing-Near-Memory (PNM) architecture that serves as the primary hardware target for the programming model proposed in this work. As illustrated in Figure 2.1, an UPMEM module complies with the standard DDR4-2400 DIMM form factor. However, conventional DRAM dies are replaced by PIM-enabled dies. Each DIMM contains up to two ranks, where each rank has 8 PIM chips. As each chip has 8 DRAM Processing Units (DPUs), there are 64 DPUs per rank and 128 per DIMM.

Each DPU exclusively manages a 64 MB DRAM bank known as Main RAM (MRAM). In addition to MRAM, the DPU contains a 24 KB SRAM block for in-

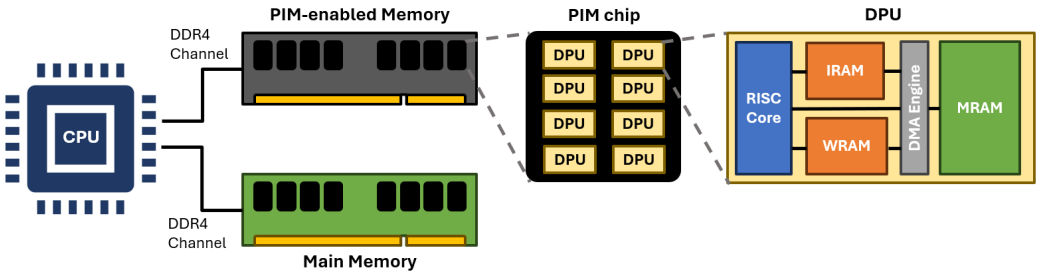

Figure 2.1: UPMEM PIM System. UPMEM PIM-enabled memory resides in a separate DDR4 channel from the host memory. Each PIM chip has 8 DRAM Processing Units (DPUs). Within each DPU is a RISC core that has both instruction RAM and working RAM (for staging data) and a DMA engine for copying data to and from the main DRAM bank.

struction memory (IRAM), and a 64 KB SRAM block for working memory (WRAM). While the host CPU can directly access the MRAM to transfer data to and from traditional host memory, it cannot access the WRAM or IRAM. Instead, WRAM serves as a local, single-cycle scratchpad for the DPU. Before the DPU can compute on any data, the data must be explicitly moved from MRAM to WRAM via DMA transfers. Furthermore, the IRAM's 24 KB capacity strictly caps the size of compiled computational kernels to 4,096 instructions, inherently limiting the complexity of algorithms that can be offloaded.

The DPU itself is a multithreaded, 32-bit, in-order RISC core that operates at a frequency of 350 MHz and has a 14-stage pipeline. However, because only the final three stages can execute in parallel, consecutive instructions from the same thread must be dispatched 11 cycles apart. To circumvent these pipeline stalls and saturate the core's throughput, the hardware supports fine-grained multithreading (up to 24 hardware tasklets). Consequently, at least 11 active threads are required to maintain maximum pipeline utilization [52].

The UPMEM instruction set architecture natively supports single-cycle 32-

bit integer addition and subtraction, but lacks dedicated hardware units for complex arithmetic. As a result, integer multiplication, integer division, and all floating-point operations require software emulation. Finally, the architecture entirely foregoes direct hardware communication between DPUs. Any inter-DPU coordination, such as reductions or global synchronizations, must be explicitly orchestrated by the host CPU via MRAM transfers. These arithmetic and communication constraints highlight the necessity of higher level programming models.

2.2 Programming Models for PIM Architectures

Programming models for these architectures can be distinguished by what the interface enables. At one end, the programmer directly targets the physical hardware. Here they remain responsible for the data transfer to processing units, breaking down the computation for these units and organizing the data to fit within the appropriate physical locations in memory. At the other end, the programmer names an operation and the system manages everything that happens below this. While most prior work exists on this end, approaches differ in terms of which operations are named.

2.2.1 Accelerator Programming Model

The programming model for many PIM architectures mirrors the discrete accelerator model from GPU programming in CUDA and OpenCL [53, 54]. In this model, computation is split between a host binary which orchestrates the work and a separate device binary for performing the computation. Here, the host and device do not share an address space and communicate through explicit data transfers across a bus.

Under this model, the host processor acts as the sole sequencer in the system. Rather than automating data orchestration, the model forces the programmer to explicitly manage the pipeline. As illustrated in the host and device routines of Figure 2.2, the programmer must write directives for:

- Dividing the input across the available processing units and determining the static memory layout of each partition.
- Managing the mechanics of the memory interface, which often requires padding partitions of unequal size and de-interleaving host data before it is usable by an individual unit.
- Loading compiled device binaries onto the units and manually triggering execution.
- Retrieving partial results from the isolated processing units and combining them on the host.

In this programming model the hardware capabilities are directly exposed to the programmer which then forces them to be responsible for both leveraging the capabilities but also managing the limitations. In systems like UPMEM without a cache hierarchy, the movement of data between a unit's local DRAM and its scratchpad is itself part of the program. The programmer must explicitly size tiles against a scratchpad measured in tens of kilobytes and manage the latency hiding of these copies. However, UPMEM systems are constrained by the IRAM capacity which limits the size of the compiled kernel. This makes more complex algorithms difficult to express easily and can require the programmer to either reload new binaries between phases or fuse them into a single, heavily branched kernel to fit within the memory limits [55, 56].

```c
/* host */
dpu_alloc(NR_DPUS, NULL, &set);
dpu_load(set, "kernel", NULL);

partition(A, parts, NR_DPUS);
pad_to_max(parts, NR_DPUS);

i = 0;
DPU_FOREACH(set, dpu, i)
  dpu_prepare_xfer(dpu, parts[i].buf);
dpu_push_xfer(set, DPU_XFER_TO_DPU,
              "mram_in", 0, sz, DEFAULT);

dpu_launch(set, DPU_SYNCHRONOUS);

DPU_FOREACH(set, dpu, i)
  dpu_prepare_xfer(dpu, res[i]);
dpu_push_xfer(set, DPU_XFER_FROM_DPU,
              "mram_out", 0, rsz, DEFAULT);

norm = reduce(res, NR_DPUS);
```

```c
/* device */
__mram_noinit uint8_t mram_in[SZ];
__mram_noinit uint8_t mram_out[RSZ];

int main(void) {
  unsigned t = me();
  if (t == 0) mem_reset();
  barrier_wait(&bar);

  T *tile = mem_alloc(TILE);
  for (o = t*TILE; o < SZ; o += NT*TILE) {
    mram_read(&mram_in[o], tile, TILE);
    compute(tile);
    mram_write(tile, &mram_out[o], TILE);
  }
  barrier_wait(&bar);
  return 0;
}
```

Figure 2.2: The accelerator model in outline. The host partitions, pads, transfers, launches, gathers, and reduces; the device kernel stages tiles between its local DRAM and scratchpad by hand. Every data-movement decision in both listings is written by the programmer.

As illustrated in Figure 2.2, the code snippet exposes the split between the host and device program for the UPMEM system. On the host side, the program acquires hardware contexts (dpu_alloc), manually partitions and pads the dataset to satisfy parallel memory interfaces, orchestrates explicit bidirectional DMA transfers (dpu_push_xfer), and then performs the final global reduction on the CPU because the individual processing units cannot communicate with one another. On the device side, the absence of a conventional hardware cache forces the programmer to manage the memory hierarchy by hand. The device kernel must manually allocate a fixed-size scratchpad buffer (mem_alloc(TILE)) and construct a precise loop to stage data between the local DRAM and the compute core using explicit blocking reads and writes (mram_read and mram_write). From this, it is clear how the program

must be written in a manner that is aware of the hardware in the system and that manually handles the data orchestration to perform the correct computation efficiently.

Aside from UPMEM, other PIM systems also provide this accelerator programming model. For the SK Hynix Accelerator-in-Memory (AiM) [57], the key operation for leveraging this architecture is a matrix vector multiplication operation. In order to directly leverage this, programmers must familiarize themselves with 4 bank activation commands, 7 compute commands, and 7 data commands along with setting the mode register appropriately. As this hardware is dedicated for matrix vector multiplication, these commands differ dramatically from the UPMEM system and utilizing this hardware would require familiarizing oneself with the microarchitectural details of the system. However, as discussed in the next section, these operations are available using a library-based programming model.

The accelerator programming model enables the programmer fine grained control over the system to fully utilize the hardware. However, by exposing low-level hardware details, this model is difficult for non-hardware experts to efficiently leverage this model to target their own applications. The programmer is given DMA calls, kernel loads, and a static assignment of data to units, but no construct with which to express a policy for orchestrating them. Writing a fast program under this model is requires manual, input-dependent co-design of the partitioning, the tiling, the phase structure, and the host-device interaction, and because each vendor supplies its own toolchain and its own transfer primitives, that effort transfers poorly from one PIM target to the next.

2.2.2 Library Model

On the other end, some PIM programming models target no code modification at all. The library model achieves this by hiding the PIM software stack entirely behind traditional, widely adopted domain-specific APIs. Beneath the library frontend, there exists a set of custom, highly optimized microkernels designed specifically for the target PIM hardware.

This approach has gained significant traction in the machine learning domain. For example, Samsung's HBM-PIM integrates its software stack directly into frameworks like PyTorch and TensorFlow [33]. The PIM runtime consists of a preprocessor, memory manager, and executor. The preprocessor analyzes the application's computational graph and identifies specific operations—such as vector addition or matrix multiplication—that are suitable for PIM acceleration. When these operations are identified, the memory manager maps the operands to the PIM memory space, and the PIM executor seamlessly configures and invokes the precompiled microkernel. Also as discussed in the previous section, SK Hynix's AiM architecture is also available as an extension to PyTorch for machine learning applications which then enables mapping certain memory-bound DNN operations to the AiM system [57].

Similar library-driven approaches have been explored for sparse linear algebra and graph processing. Frameworks like GraphPIM [58] intercept standard graph API calls and offload memory-bound atomic operations to PIM instructions, while PIM-enabled BLAS libraries (Basic Linear Algebra Subprograms) allow legacy scientific applications to benefit from near-memory acceleration simply by dynamically linking to a PIM-aware mathematical backend [59].

However, this abstraction is limited in terms of expressiveness and flexibility.

The library model treats these operations as a black box. If an algorithm relies on a computational pattern or a composite operation that has not been explicitly implemented by the library developers, the runtime cannot accelerate it. The library model trades off usability for strict constraint on expressible applications. To enable domain experts to leverage memory-centric hardware for novel algorithms, a programming model should provide a middle ground. The abstraction should be high-level enough to hide explicit hardware orchestration, yet expressive enough to dynamically support arbitrary, user-defined operations over regions of memory.

2.2.3 Summary

In summary, the existing programming methods for Processing-In-Memory architectures expose a fundamental tension between expressiveness and abstraction. The accelerator model and the library model demonstrate the two ends of this spectrum but neither is sufficient for general-purpose algorithmic research on memory-centric systems.

The accelerator model where the hardware is programmed directly provides the most flexibility by enabling general-purpose computation by exposing the bare-metal hardware to the programmer. However, this approach fails to provide any abstraction and the programmer is forced to act as the system orchestrator. This means the programmer must explicitly manage memory partitioning, data movement between the host CPU and PIM-enabled memory, and manage communication between isolated banks of memory. Consequently, algorithms become tightly coupled to the microarchitecture of a specific vendor's hardware and tools which prevents portability and places a large engineering burden on domain experts.

Conversely, the library model swings to the opposite extreme, providing ab-

straction without flexibility. While it allows programmers to easily leverage PIM acceleration through familiar frameworks without managing hardware specifics, it restricts them entirely to a rigid, pre-defined set of supported operations. When a novel algorithm or a custom mathematical operation falls outside this supported set, the programmer is left completely unable to use PIM acceleration even if the operations described can be performed by the accelerator.

Ideally, new layers of abstraction must be introduced to decouple arbitrary algorithm descriptions from their hardware implementations. Rather than forcing the programmer to write manual data-movement schedules or trapping them in rigid libraries, a memory-centric programming model should allow developers to declaratively specify the computation to be performed over regions of memory. By doing so, the system itself can dynamically manage the placement and movement of data based on the underlying hardware constraints.

To design and evaluate such a declarative model, it is necessary to first understand the algorithmic targets that stand to benefit most from near-memory acceleration. Because PIM systems are uniquely positioned to alleviate data-movement bottlenecks, memory-bandwidth-bound iterative algorithms serve as ideal candidates. The following section introduces sparse linear solvers which are a class of algorithms deeply bottlenecked by memory bandwidth. These algorithms which will serve as the primary workloads for evaluating the programming models proposed in this dissertation.

2.3 Sparse Linear Solvers

In this dissertation, the focus remains on sparse linear solvers given the prevalance in many modern applications. This class of applications possess a wide range of

approaches but this work focuses on adapting some of the more common approaches. However, in order to follow this later on, some requisite background is needed on solvers in this space.

Solving a system of linear equations can be represented by the formula $\mathbf{Ax} = \mathbf{b}$ where

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

and A and b are known and x is the variable that is being solved for. This formulation arises in a wide range of domains. In structural mechanics, discretizing a solid creates a stiffness matrix which naturally leads to sparsity as elements only interact with their neighbors. In PDE simulations, a common reference is HPCG (high performance conjugate gradient), which solves a poisson equation on a regular 3D grid. In these applications the A matrix is often sparse and could scale to be extremely large. Here the dimensions could be on the order of millions of elements.

Solving this formula can be done using various types of iterative methods. These iterative methods vary in terms of the convergence rate and the the types of matrices that they are best fit for. For these approaches an initial guess of $x^{(0)}$ is often used and $x^{(k)}$ is used to denote the k-th approximation of x. Convergence is often defined by the changes being made per iteration being below some tolerance or a sufficiently small residual where the residual, r, is defined by $r = \left\| \mathbf{b} - \mathbf{Ax} \right\|$.

Next, a few basic approaches for solving these linear systems are described. Understanding these approaches is necessary in order to understand how these approaches can be co-designed with the hardware constraints as described later in

Section 4.3.

2.3.1 Jacobi Iteration Method

The Jacobi method is an iterative algorithm that can determine the solution for a strictly diagonally dominant system of linear equations. In each iteration, the diagonal element is solved for and the appropriate value is plugged in. Using matrix form, the matrix $\mathbf{A}$ is decomposed into $\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$ where

$$\mathbf{D} = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}, \mathbf{L} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}, \mathbf{U} = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

.

The solution is then obtained iteratively with

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{x}^{(k)})$$

or per element this is

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)})$$

Here, convergence is determined by the spectral radius, ρ, of the iteration matrix where iteration matrix is $\mathbf{J} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$. As different types of iterative methods are used later, this property will be important in order to aid convergence.

This computational pattern lends Jacobi to be embarrassingly parallel as each output element can be computed as an independent operation. However, parallelism and data movement are important to enable PIM adoption. While this algorithm

requires no interbank communication to compute an output, this operation still requires streaming the entire matrix during every sweep. When $\mathbf{A}$ is sparse, the data movement per iteration is now $d = 4(n+1) + 4 * nnz + 8 * nnz + 3 * 8 * n$ where d represents the data movement per iteration in bytes, n is the dimension size of the matrix and nnz is the number of nonzeros in the A matrix. Here, the bandwidth bottleneck is not addressed by naively mapping this application to PIM hardware.

2.3.2 Gauss-Seidel

Gauss-Seidel is another common iterative method that is used to solve a system of linear equations. This method is only guaranteed to converge if the matrix is either strictly diagonally dominant or symmetric positive definite. For this method, the $\mathbf{A}$ matrix is decomposed as $\mathbf{A} = \mathbf{L} + \mathbf{U}$ where

$$\mathbf{L} = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \mathbf{U} = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

and per iteration the update is computed via:

$$\mathbf{x}^{k+1} = \mathbf{L}^{-1}(\mathbf{b} - \mathbf{U}\mathbf{x}^{(k)})$$

which leads to the following computation per iteration:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)})$$

Computationally, this method is similar to the Jacobi method but with the distinction that the computation of $x^{(k+1)}$ uses elements of $x^{(k+1)}$ that have already

been computed. This long dependent chain of operations thus makes this approach more difficult to parallelize. Here the iteration matrix is given by $\mathbf{M^{-1}N}$ which leads to difference convergence rates for this approach versus Jacobi iteration.

2.3.3 Successive Over-Relaxation

This method is a variant of Gauss-Seidel that introduces a constant ω which serves as the relaxation factor. This constant is used to enable faster convergence and can be used for any slowly converging iterative process. In matrix form, $\mathbf{A}$ is decomposed into $\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$ again and now the iteration in matrix form is as follows:

$$\mathbf{x}^{k+1} = (1-\omega)\mathbf{x}_i^{(k)} + \omega\mathbf{D}^{-1}(\mathbf{b} - \mathbf{L}\mathbf{x}^{(k+1)} - \mathbf{U}\mathbf{x}^{(k)})$$

The new element-wise update formula is now:

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)})$$

In order for this method to converge for any initial guess, the relaxation factor must fall within the range $0 < \omega < 2$ based on Kahan's theorem [60]. Then if these criteria are met, if the matrix $\mathbf{A}$ is symmetric and positive definite then this approach is guaranteed to converge by the Ostrowski-Reich theorem [61]. As the Filter-Update programming model introduced in the next section modifies the iteration matrix in each of these iterative solvers, this technique of introducing a constant ω is essential to achieve convergence within a comparable number of iterations.

2.4 Summary

The iterative solvers discussed in this chapter are fundamentally bottlenecked in modern applications by the amount of data movement that is required. As domain scientists continue scale their applications to solve larger sparse linear systems, the requirement to stream the entire matrix and solution vector across the memory bus in every iteration becomes prohibitively expensive. For this reason, moving some computation to near memory hardware to eliminate or reduce the data movement required serves as a promising solution. However, for domain experts to both develop new approaches for solving these systems and leverage this new hardware, a programming model that allows developers to reason against higher level operations without managing underlying microarchitectural complexities is needed. Here the thesis is that a declarative specification of high level operations that can be efficiently mapped to near memory hardware described upon regions of memory is the appropriate level of abstraction. As the following chapters will demonstrate, a specification on regions of memory provides the system sufficient information to efficiently map computation to near-memory hardware and grants programmers the flexibility to develop new data-movement efficient algorithms that leverage these operations.

Chapter 3

Filter Update Programming Model

This chapter provides an overview of the Filter Update programming model from the specification of the computation down to the hardware. This chapter illustrates both the API for the programming model and the compute stack necessary for lowering these operations to potential hardware targets in memory-centric systems. The details for how filter and update operations are run on near memory hardware and the corresponding performance on real PIM hardware is demonstrated here. Furthermore, future potential PIM architectures that support filter and update operations are outlined.

3.1 Filter-Update Overview

For memory-centric systems where near memory computation has limited computation ability, filter and update serve as two key operations for both performing useful computation that can be done on the algorithm side and being types of com-

putation that can benefit from near memory computation. While the algorithmic requirements will be discussed in Chapter 4, the design of the programming model is highlighted here.

The filter operation maps a predicate to near memory hardware. This predicate determines whether an element is to be transferred to the host. Following this the output of this operation is in either one of two formats. One possible format is only the indices of the elements that survived are transferred to the host while the other is both the indices and filtered value are transferred. How this computation is mapped to the near memory hardware present in the system is thus not exposed to the programmer. Actual hardware is targeted by lowering these high level operations through the filter update compute stack discussed in the next section. Here the abstraction focuses upon data within a region of memory. The information that is exposed about this region is the data type, number of elements and each element resides at a certain index (that is less than the number of elements and nonnegative) within the region of memory. For the large datasets that this operation is designed for, the cost of this operation depends upon the data that is selected rather than the size of the entire dataset. Here the bottleneck remains the memory bandwidth as the arithmetic operations remain inexpensive and near memory, higher internal bandwidths can be achieved than when compared to the bus between CPU and DRAM. This result is highlighted further later in Section 3.3.1.

The update operation accumulates the data which is passed in using a coordinate list with the data within a given memory region. As each element in a memory region has a data type and every element has a corresponding index, the indices of the data that are passed in are used to align the data with the appropriate element within the memory region. The operator that is specified is used to

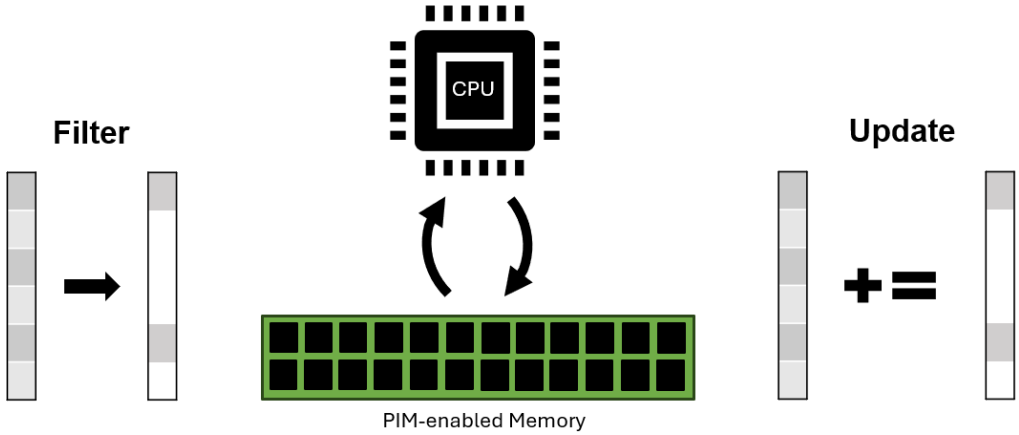

Figure 3.1: Illustration of data movement in the filter-update programming model. PIM-enabled memory is used to evaluate basic filter and update operations while the traditional host CPU is used to perform more complex arithmetic operations which do not work well in memory like multiplication and floating point arithmetic.

accumulate the value with the resident data in the form $a = a \bigoplus b$ where b is the value that is passed in. The valid operations here are technically dependent upon the system, the operation as specified does not necessarily map directly to a single instruction or operation in near memory hardware. One such example of this is highlighted later as floating point subtraction is performed on hardware that does not natively support floating point operations.

While taken in isolation these operations do not appear to do very much, however the key insight here is that the complex arithmetic operations common in many of these iterative algorithms cannot be efficiently mapped to this near memory hardware. Scientific applications typically require floating point arithmetic along with multiplication and division operations. While these are standard mathematical operations, they are difficult to implement in hardware and thus are uncommon in many PIM architectures. For compute that is placed near memory, tight area and power constraints prevent adding any large arithmetic logic units (ALU) or floating

point units (FPU). For this reason, this programming model prescribes that these operations remain on the host CPU. As illustrated in Figure 3.1, the PIM enabled memory is only used to perform these two operations. Each iteration begins with a filter in memory which leads to a coordinate list that is sent to the host. The host then computes the correction without restrictions on multiplication or floating point operations. Then the update with the coordinate list is issued and the memory accumulates these values. Following this the cycle repeats. Here the improvement in performance and energy comes from reducing the amount of data that is transferred across the memory bus. By performing the filter operation in memory, an important subset of the data can be selected and can thus compute an update that is more significant compared to randomly selected subsets or smaller contiguous blocks from memory. This difference is illustrated in Chapter 4 where filter-update modified approaches are evaluated.

In order to map these operations to PIM hardware a compute stack must be developed. This enables programmers to utilize a high-level API while the computation is lowered down to the actual hardware targets present in the system. The design of the filter update API and compute stack are described in the next section.

3.2 Filter Update Programming Model Compute Stack

To effectively program novel PIM architectures, a compute stack must be designed to bridge the semantic gap between high-level algorithmic intent and low-level hardware constraints. As established with the accelerator programming model, directly programming capability-limited near-memory hardware forces the developer to manually orchestra data partitioning and alignment along with invoking the kernel appropriately. This creates hardware-specific implementations that prevent domain

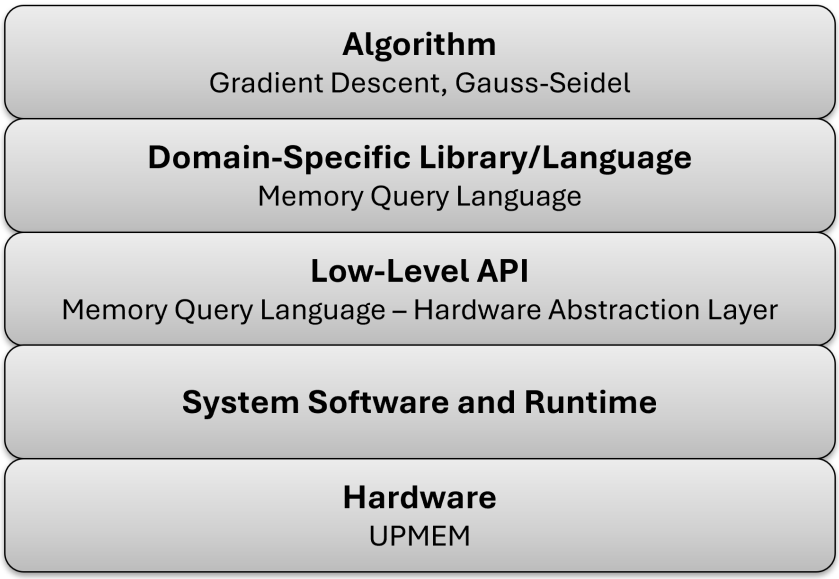

Figure 3.2: PIM Compute Stack

experts from easily exploring new numerical methods.

To manage this complexity, the filter-update programming model utilizes a compute stack as seen in Figure 3.2 to systematically lower the computation from the mathematical algorithm design down to the physical execution on the hardware. At the highest level, iterative numerical methods and optimization algorithms are formulated using foundational mathematics. Here, the focus is purely on data analysis and convergence properties. Following this, a domain-specific language is used to provide a high-level abstraction for domain scientists to express the program. Here, the Memory Query Language (MQL), inspired by database queries is introduced. Like database queries which are agnostic to how the data are physically stored, MQL allows for a declarative programming paradigm to allow users to describe computation upon logical regions of memory. To transition from a hardware agnostic specification to an executable, MQL is lowered to the MQL Hardware Abstraction Layer (MQL-HAL) which acts as a virtual set of instructions for targeting

an arbitrary, abstract PIM device that exposes the key in-memory compute operations. Finally, the MQL-HAL is lowered to the system software that is actually present in the system. This could range from a PIM simulation framework to the vendor-specific runtime and drivers necessary for executing the filter and update operations on PIM hardware.

As Chapter 4 focuses in depth on how algorithms can be designed for the Filter-Update model, the next section focuses on the next level of the PIM compute stack, the Memory Query Language (MQL).

3.2.1 Memory Query Language (MQL)

To ensure portability across different PIM devices, the Memory Query Language (MQL) isolates the application description from the low-level data representation and computation details. Inspired by SQL, MQL utilizes a declarative programming model where users select regions of memory (instead of tables) and describe the computation to be performed on the data at those addresses. The system captures these semantics in an abstract syntax tree, which can then be used to target a code generation system like SPIRAL [62] to generate instructions for a specific PIM architecture.

While PIM systems allow bandwidth-limited algorithms to exploit high internal DRAM bandwidth, a method is needed to efficiently handle the sparse results that must be read or written by the CPU. MQL solves this by introducing two operations: *filter*, which outputs sparse data from a dense region, and *update*, which applies a sparse input to a dense region.

Filter Operation. As illustrated in Figure 3.3 the filter operation can view a region of memory as similar to that of a table. This table has two attributes, the

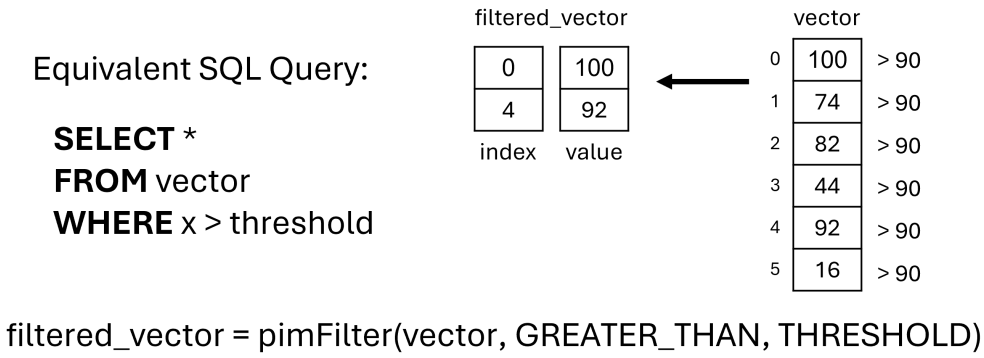

Figure 3.3: Equivalent SQL query for a filter operation.

index of the location of each element and the value at that location. This operation is then equivalent to that of a filter predicate from database applications. Every element within a region of memory is scanned and the predicate that is passed in determines whether the element is included. What the predicate is able to reference here is dependent upon the near memory hardware. For unary predicates (e.g.: $x > \tau$), this maps well to even basic bit-serial architectures as this results from a simple comparison against a constant value that is passed in. Binary predicates (e.g. $x[i] > y[i]$) or predicates that require values from other regions of memory or are dependent upon other values within the same region of memory are not permitted. This is due to the physical placement and split of a "memory region" across multiple processing units when in near memory computation. As such, communication between processing units is often expensive or not possible as described in the discussion on PIM architectures in Chater 2. Conjunctions or ranges (e.g. $\tau_1 < x < \tau_2$) are possible as this is only requires one additional constant and comparison operation when compared to the unary operation. Predicates that involve arithmetic on the element (e.g. $\alpha x > \tau$) are generally not allowed as multiplication cannot be performed efficiently on many PIM architectures. Lastly, predicates that involve

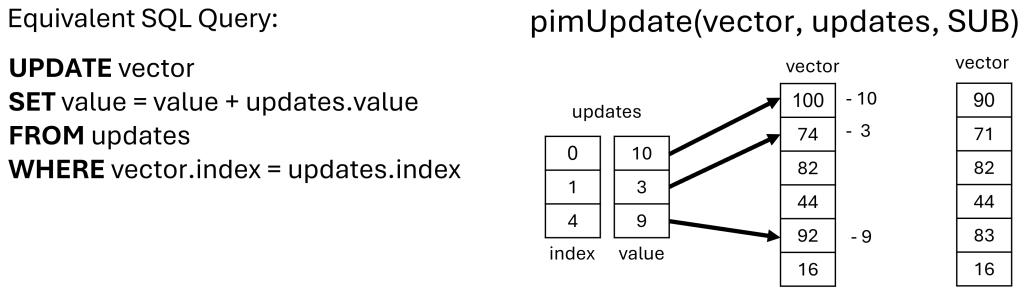

Figure 3.4: Equivalent SQL query for an update operation

global state, like top k% of elements or the largest element, are not possible as this requires state from across all elements within a memory region. For algorithms like Gauss-Southwell [39], this is the traditional approach for selecting the subset of elements to use within an iteration, However, as shown Chapter 4, these approaches can be adapted to use filter operations that are more amenable to near memory acceleration.

Update Operation. The corresponding SQL query for an update operation can be seen in Figure 3.4. When compared to traditional operations in database applications, it is similar to that of group-by aggregation. The region of memory is still viewed as a table with an index and value and now the grouping key is done by index and the aggregate function is the operator that is passed in. Here the operation is simpler than the database equivalent as the number of indices is already known thanks to the fixed size of the memory region and each element will only be accumulated with one other input. The operator here is once again dependent upon the hardware. Most PIM hardware can perform some basic integer add operation or other binary operators which can be used as the accumulation operator. Maximum or minimum operations are also relatively cheap to implement near memory as this only requires a single comparison operation (which has already been established

from the filter operations). Here, each index appears at most one time for a valid input to the sparse input to the update operation. For this reason and as the update vector is always used as the second operand, operations like integer or floating point subtraction are also valid. While integer addition operations are straightforward in many near memory architectures, floating point is not necessarily a given. How floating point addition and subtraction can actually be performed in near memory hardware is described later as this high level operation is lowered to hardware.

3.2.2 MQL Hardware Abstraction Layer

To achieve good performance, high-level MQL commands are implemented through the Memory Query Language Hardware Abstraction Layer (MQL-HAL). MQL-HAL targets an abstract PIM device, exposing pertinent hardware details to the program. In addition to filter and update operations, this layer introduces an API for promoting and demoting data to PIM-enabled memory. In Listing 3.1, an example MQL-HAL program can be seen for a modified gradient descent algorithm. This listing demonstrates how the filter and update operations target an abstract PIM device while the rest of the computation on the host side is simply able to utilize the sparse output and input from these operations to perform the appropriate computation on the host CPU.

Table 3.1: Filter predicates provided by the hardware abstraction layer.

Predicate	**Condition**
`GREATER_THAN`	$x > \tau$
`LESS_THAN`	$x < \tau$
`ABS_GREATER_THAN`	$\|x\| > \tau$
`USER_DEFINED`	Supplied by the programmer.

At the MQL-HAL level, the filter and update operations require specific pa-

```
CSR filtered_grad, filtered_grad_update; // Host side CSR struct
for (int iter = 0; iter < N_ITERATIONS; iter++) {

    filtered_grad = pimFilter(gradient_pim, ABS_GREATER_THAN,
    THRESHOLD_VAL);

    // Host side update computation
    filtered_grad_update.nnz = filtered_grad.nnz;
    filtered_grad_update.keys = filtered_grad.keys;
    filtered_grad_update.values = (GD_Precision*)malloc(filtered_grad.
    nnz * sizeof(GD_Precision));
    for (int i = 0; i < filtered_grad.nnz; i++) {
        filtered_grad.values[i] *= LR;
        filtered_grad_update.values[i] = filtered_grad.values[i] *
    gd_coeffs[filtered_grad.keys[i]];
    }

    pimUpdate(x_pim, filtered_grad, SUB);

    pimUpdate(gradient_pim, filtered_grad_update, SUB);

    free(filtered_grad_update.values);
}
```

Listing 3.1: MQL-HAL program for filter-update modified gradient descent.

rameters, such as the size of the memory region, to expose the necessary hardware details. The filter operation applies a predicate to each element in a PIM memory region, outputting those that satisfy the condition in a sparse data structure. Common operations (e.g. greater than, less than, equals) are provided as built-in functions, but user-provided functions could be passed in using a library that serves as a specification for the computation to be performed in PIM. This filter operation takes in the following arguments:

1. A target PIM memory region
2. A sparse data structure that defines the output format, e.g. index-only, index and data, and compressed sparse row (CSR).
3. A predicate in the form of a stateless function that takes in a single value from a memory address as input.

The list of valid filter predicates can be seen in Table 3.1. While these demonstrate a few common operations that map well to most PIM architectures, they are not necessarily an exhaustive list. For this reason, a user defined predicate is still able to be supplied for PIM architectures that have different capabilities.

The update operation modifies values in PIM-enabled memory by applying a sparse data structure. The elements of the sparse input are matched to the corresponding dense PIM memory region by their index. In the sparse data structure, the index of each piece of data is passed in where as in the dense data structure the index is determined based on the position of the data in memory. This process is defined by:

1. A target PIM memory region

2. A sparse input data structure

3. A binary update function that takes an element from the sparse data structure and a corresponding element from the dense PIM memory region to produce an output.

Like the filter operation, the update operation can use both built-in functions and user-provided functions to define the update function as seen in Table 3.2. While the user defined operator provides some room for flexibility, it is important to note that the update operation must remain computationally inexpensive in order for the update operation to be used effectively. Without dedicated support in hardware, using operators like multiplication or division should not be performed.

Table 3.2: Update operators provided by the hardware abstraction layer.

Operator	Function
`ADD`	$a \leftarrow a + b$
`SUB`	$a \leftarrow a - b$
`REPLACE`	$a \leftarrow b$
`USER_DEFINED`	Supplied by the programmer.

Beyond the filter and update operations, this level possesses more common operations for managing data movement to PIM-enabled memory and using PIM hardware. The full set of operations can be seen in Table 3.3. Here we have operations for abstracting device specific function calls for initializing and deinitializing PIM hardware along with operations for PIM memory allocation. At this level, PIM-enabled memory is not treated the same as host memory due to the constraints of many systems. For this reason, there are specific memory allocation functions for using PIM memory along with functions for copying data to and from PIM enabled memory.

Table 3.3: Operations provided at the hardware abstraction layer. Bracketed arguments are optional.

Operation	Arguments	Description
Initialization		
`init_pim`	`[n]`	Initializes the PIM backend and reserves the processing units to be used.
`shutdown_pim`	—	Releases the processing units and any remaining device allocations.
Region management		
`pimAlloc`	`num, data_type`	Allocates a region of `num` elements of type `data_type` in PIM-enabled memory.
`promote2PIM`	`mem, num, data_type`	Places the host array `mem`, holding `num` elements of type `data_type`, into PIM-enabled memory, returning a PIMRegion that refers to it.
`pimCopy2Host`	`region`	Copies the contents of `region` back to its host-accessible memory.
`pimFree`	`region`	Releases `region` and its device allocation.
Computation		
`pimFilter`	`dest, type, threshold, [max_updates], [start_idx], [keys_only], [end_idx]`	Evaluates the predicate `type` against `threshold` over `dest` and returns the elements that satisfy it as a coordinate list. The scan may be bounded to the subrange [`start_idx`, `end_idx`), capped at `max_updates` results, and, with `keys_only`, restricted to indices alone.
`pimUpdate`	`dest, csr, type`	Accumulates the coordinate list `csr` into `dest`, aligning each incoming value with the resident element at its index and combining the two with the operator `type`.

MQL-HAL is then further lowered to a PIM API which compiles to target certain PIM architectures. For user-provided functions, tools like LibraryX [63] can perform semantic capture and generate a dataflow graph of the computation. This representation can then be used by the SPIRAL [62] code generation system for analysis and optimization. For supporting new PIM architectures, this low-level API simply needs to be reimplemented using the new PIM target's ISA.

3.2.3 System Software and Runtime

Below the Hardware Abstraction Layer (MQL-HAL), the system software and runtime are responsible for lowering abstract data structures and operations into device-specific API calls. While the MQL-HAL exposes generic memory regions and computation types (like filter and update), the runtime translates these into the exact microarchitectural instructions required by the underlying hardware. This layer is where the vendor-specific and device specific operations now lie.

While this level of abstraction differs depending upon the system there are a few operations that remain common. One of these is that the runtime is responsible for managing physical memory allocation in PIM-enabled memory. When the MQL-HAL allocates a PIM-enabled region, the runtime translates this request into device-specific initialization calls. On the UPMEM system, instead of the programmer explicitly allocating hardware contexts (e.g., calling `dpu_alloc` to reserve a set of DRAM Processing Units (DPUs)), the runtime evaluates the available hardware and provisions the device. It handles the subsequent memory allocation and deallocation routines mapping logical memory regions to physical addresses on the device (such as UPMEM's MRAM) and ensures that data transferred from the host is correctly interleaved or formatted for the target architecture.

Focusing further on what this looks like for the UPMEM system, when a dense vector is allocated, the runtime distributes the memory region across the processing units reserved when the device is initialized. Rather than forcing the programmer to manually calculate tile sizes and pad data structures, the runtime automatically divides the memory region into equally sized contiguous chunks. For example, a vector of size N distributed across D active DPUs will be partitioned into chunks of size $\lceil N/D \rceil$. The runtime is also responsible for orchestrating the

execution of the filter and update operations. A single device binary is loaded when the system is initialized and is capable of performing each operation described by the filter update model. When an operation like `pimFilter` or `pimUpdate` is invoked, tasks corresponding to the operation carrying the function arguments (e.g. predicate and threshold or data to accumulate) is dispatched and then launched on the DPUs. The details of how this computation is performed on the hardware are described later in Section 3.3.1

3.2.4 Filter Update Operations on PIM Architectures without Floating Point Support

As PIM hardware is heavily constrained in terms of the area and power constraints of the logic that can be placed in memory, these compute units frequently lack support for floating point or double precision computation. On the UPMEM architecture, DPUs do not have a floating point unit and possess no native integer multiplication or division instructions either [64]. These operations are compiled into a shift and add algorithm where an operand is broken down into powers of 2 and then left bit shifting and addition is used to perform the appropriate operations. However, as floating point computation is essential for many scientific applications, a new approach must be taken in order to utilize PIM hardware for these algorithms. In soft float [65] floating point number representations are split into the mantissa and exponent bits. For 64-bit doubles the 53 bit mantissa is stored in a two's complement representation in a 64 bit integer and the 11 bit exponent is stored in a 16 bit integer. Following this double precision computation can be performed using integer hardware using bitwise operations. The extra bits in the 64 bit integer that is used to store the mantissa provide the headroom to defer the rounding mode logic

until necessary. This split between exponent and mantissa into separate integers is leveraged further for filter and update operations on the UPMEM architecture.

As opposed to other floating point formats and implementations on integer hardware, this approach is co-designed with the PIM hardware in mind. While block floating point formats such as NVIDIA's NVFP4 [66] can share the exponent bits across elements, this however ties together neighboring elements that may vary by many orders of magnitude. In addition, the representation would prevent the filter operation from being able to reduce the data that needs to be read by only looking at the exponent bits. The details of how this is performed on commercial PIM hardware are described in Section 3.3.1

3.3 Target Hardware

Modern PIM architectures are programmed in a manner that forces application developers to be aware of the PIM hardware. This largely stems from the fact that depending upon where in memory the compute unit is placed (rank level, bank level, subarray level) greatly changes the types of instructions the programmer is allowed to use.

3.3.1 UPMEM

As introduced in Chapter 2, UPMEM is a PNM architecture where the compute units are placed at the bank level.

The filter update programming model enables leveraging hardware that previously did not seem possible. UPMEM compute units do not have native floating point support which ostensibly makes mapping scientific applications to this hardware seem infeasible. However, for basic filter and update operations, bitwise

transformations as seen in soft float can be used to efficiently perform floating point arithmetic on hardware without native integer support.

In order to enable using the UPMEM hardware for various vectors that need to be filtered, the MRAM is statically split into 64 slots. Each of these slots is able to accommodate a PIMRegion which can be an operand for a filter or update function. This static mapping avoids the need for any dynamic allocation or operating system as the DPU possesses neither. Within a slot the data is split across DPUs such that each DPU has the elements $[i * m, (i + 1) * m)$ where i is the DPU index and m is n/D where n is the size of the vector and D is the number of DPUs. When using software emulated floating point, each data element (64 bit float) is split into a 16 bit exponent and 64 bit mantissa. The exponent bits for each element are stored in the first fifth of the slot while the mantissa bits are stored in the remainder. This layout is essential for optimizing filter operations for this data format.

On the UPMEM hardware, the filter operation occurs in four phases. First, the host processor issues a task that contains the slot ID that is to be filtered and the threshold value. On each DPU, each slot is split into T disjoint ways where T is the number of threads. Then, each DPU overlaps the transfer time of an independent DMA with the compute time of the filter operations where the latency of the DMA of each thread is overlapped with the compute time of the other T threads. The elements that pass the filter are then written to a per tasklet reply slot and finally, the results are written back to a host buffer. For data that is being written back, this could be index only or index and value only and this configuration is passed in by the filter operator. Here the time to transfer from MRAM to WRAM via DMA scales with $O(n/D)$ while the write back stage is more dependent upon the number of elements that pass the filter. This scan from

MRAM is bandwidth bound by the local DPU bandwidth rather than the host DRAM bandwidth. While the initial scan operation is load balanced across DPUs as they all possess the same number of elements, the resulting number of operands from the filter is not. The kernel launch is a synchronous operation but as each DPU scans the same amount of data the variability between the fastest and slowest DPU is not significant. However, on the reply all DPUs are blocked by the DPU that returns the most elements. However, as the filter operation requires no data that is resident on other DPUs and is not dependent upon the results from other threads, this remains as a cheap primitive operation that can be mapped to this PIM architecture. The filter operation actually requires sequential local reads, an element-wise predicate and a write append operation.

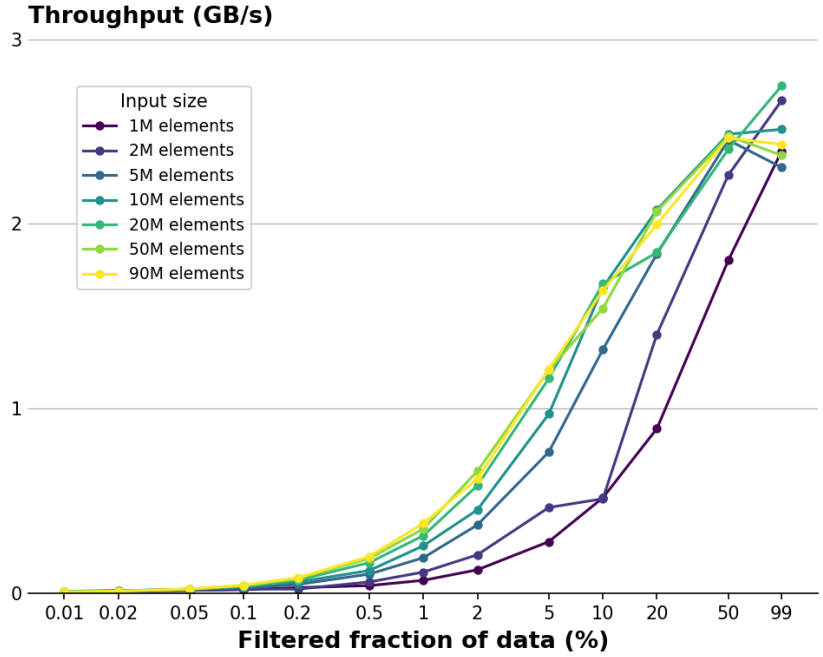

Figure 3.5: UPMEM filter operation throughput for various filtering percentages across a range of vector sizes

In Figure 3.5, the results from profiling a greater than filter can be seen. The x-axis corresponds to how much of the data in the vector is filtered and the y-axis shows the effective throughput in terms of GB/sec. The throughput here is measured by using the total amount of data transferred and the time for the total filter operation. Each color here represents a vector of a different size. From here, it can be seen that the throughput lags behind the peak bandwidth of the system. In fact, as seen in Figure 3.6, even when the amount of time spent on computation is ignored and only the time for transferring data from UPMEM to host memory is accounted for the performance still is unable to match the peak DDR4 bandwidth for the system. While this obviously provides a meaningful speedup, here the lower peak bandwidth becomes more obvious. As the amount of data that needs to be transferred is around 10MB or more, the data transfer bandwidth peaks at around 12 GB/s.

When performing the filter operation on UPMEM hardware for floating point values, some optimizations may be made in order to reduce the amount of data movement and computation that is necessary. For filter operations such as $|x_i| < \tau$, the threshold (τ) can be rounded to the nearest power of two which thus allows for only reading and performing the comparison on the exponent bits rather than reading the entire data. In order to perform floating point arithmetic on UPMEM hardware, these floating point operands have already been broken down into the soft float representation where 16 bit integers are used to represent the exponent while 64 bit integers hold the mantissa. By rounding the requested threshold to the nearest power of 2, the mantissa bits can be ignored and only the 16 exponent bits need to be read in order to evaluate the threshold. As the threshold is a power of 2 the mantissa is all 0s which thus enables only comparing the exponent bits.

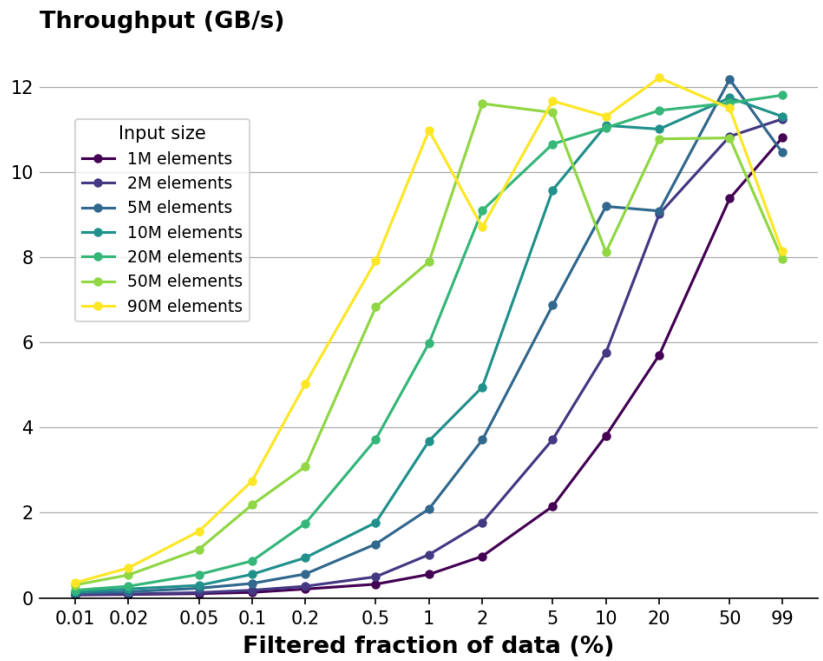

Figure 3.6: UPMEM filter operation throughput while only accounting for DPU to host DRAM communication time.

This means that when a thread returns 0 elements, this costs a read of 2 bytes per element rather 8 bytes per element in the thread.

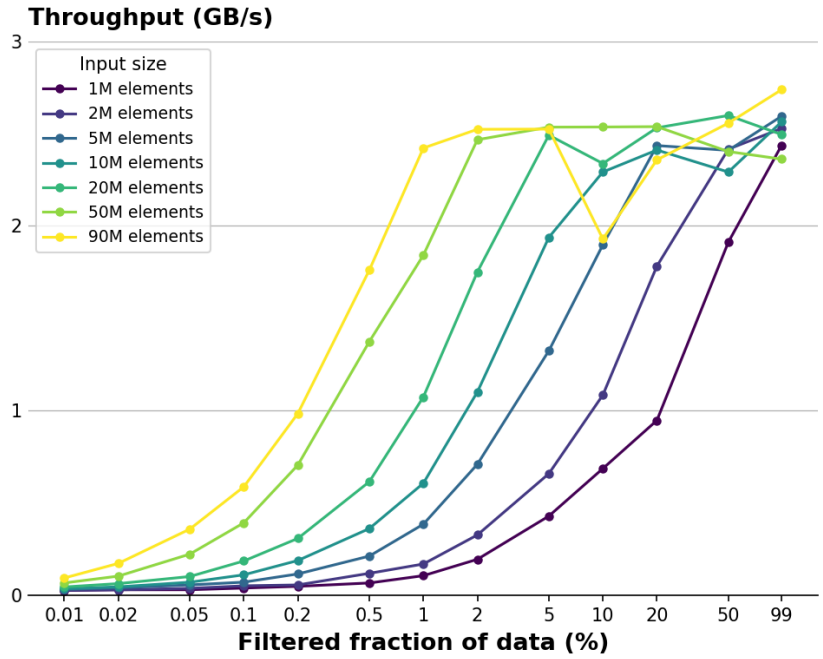

Figure 3.7: UPMEM update operation throughput for various filtering percentages across a range of vector sizes

The same analysis can be repeated for the update operation. In Figure 3.7 and Figure 3.8 the throughput for the update operation on UPMEM can be seen. Here the sparse data is being transferred from the host to the UPMEM system instead. On the UPMEM hardware a floating point addition operation is performed using the soft float technique described earlier. Here, the bottleneck at 2.7 GB/sec can be seen again. Here, it can be seen that the maximum throughput of this operation does not seem to be dependent upon the number of elements in the vector that is being accumulated to. Instead, no matter what the vector size is, the throughput of the update operation peak is based upon the number of elements that are trans-

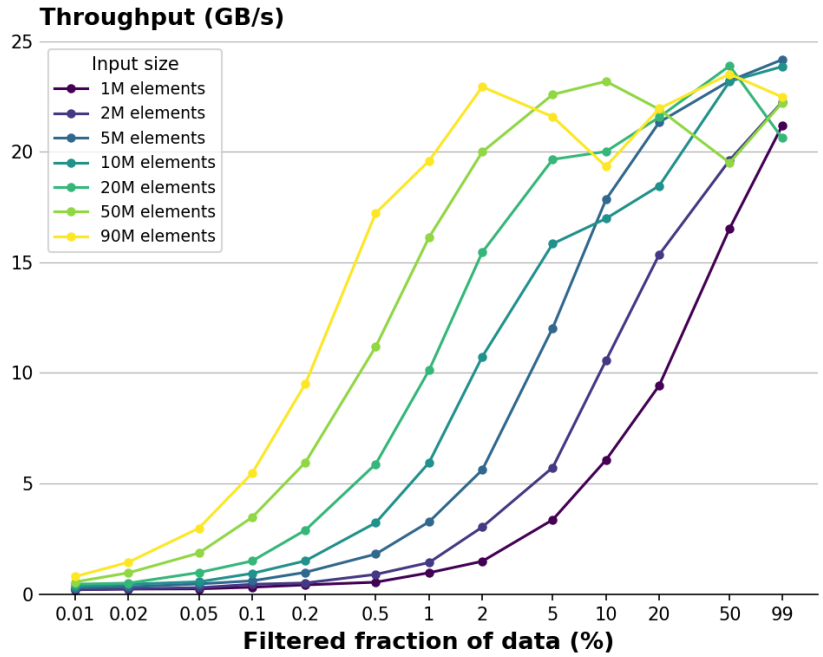

Figure 3.8: UPMEM update operation throughput while only accounting for DPU to host DRAM communication time.

ferred across the memory bus. As the number of elements to be updated grows or as the vector size grows, the throughput of the update operation does not deteriorate but rather stays constant as the bottleneck is the data transfer time.

The effects seen here are a limitation of the UPMEM system in its current form. As prior work has shown, it is not possible to achieve the full DDR4 bandwidth that the standard interface provides [67, 68]. When reading data from DRAM, the UPMEM memory controller will not take advantage of the row buffer and instead performs the row activation for each read to a row of DRAM. In addition, in order to simplify the compute near memory, the interleaving of data is undone in software using the UPMEM memory controller. While in traditional memory controllers a 64 bit data element is split into 8 bit chunks that are spread across 8 banks

and all activated simultaneously, this is not the case for UPMEM. In UPMEM this 64 bit data element is written within a single bank and requires 8 consecutive activations to memory in order to read out a single 64 bit value. As a result, when trying to read the data out from memory, the peak UPMEM bandwidth is dramatically lower. In order to get peak DDR4 bandwidth, the UPMEM memory controller that performs the interleaving must be modified. Here the tradeoff lies in the fact that no interbank communication is required in order to read a single operand to a DPU from DRAM. However, as the algorithms that seek to utilize near memory acceleration are memory bandwidth bound, reducing the amount of memory bandwidth in these systems makes it difficult to achieve performance gains. For this reason, in Chapter 4, a speed of light model representing a PIM system able to achieve performance comparable to that of the standard interface is used to estimate what the performance could be. In the next section, new potential PIM hardware is discussed that would enable taking advantage of filter update operations using the full memory bandwidth.

3.4 Summary

In this chapter, the Filter-Update programming model was detailed across the entire compute stack, from the high-level API, through the intermediate abstractions used to lower operations, down to their physical execution on near-memory hardware. By restricting in-memory computation to simple element-wise predicates and accumulations, operations can be efficiently mapped even to highly constrained PIM architectures that lack floating-point units, hardware multipliers, or inter-bank communication. By leaving complex arithmetic on the host CPU and pushing only bandwidth-intensive data sparsification and accumulation to memory, the model

efficiently utilizes capability-limited hardware.

Here the effectiveness of using a specification between regions of memory is seen from the systems perspective. The declarative specification provides enough information for the programming model which uses logical memory regions to automatically lower the computation to near memory hardware. The various levels of the compute stack how the system is able to handle data orchestration between host and PIM-enabled memory, partition data across compute units and execute the hardware kernels. In the next chapter, the effectiveness of this programming model is viewed from the algorithms perspective.

Chapter 4 explores how programmers can leverage these primitives to design highly data-movement-efficient algorithms. The Filter-Update model is practically demonstrated on both existing optimization algorithms and newly adapted filter update linear solvers. These applications serve to demonstrate how this model successfully resolves the tradeoff between flexibility and ease of use described earlier. The Filter Update programming model allows for transparent use of the PIM hardware like a library-based programming model, while preserving the flexibility required to express novel algorithms.

Chapter 4

Filter-Update Applications

This chapter investigates applications that can effectively leverage the Filter-Update model. Because these operations are optimized on PIM hardware to minimize data transfer between the host CPU and main memory, we focus specifically on algorithms bound by memory bandwidth. Iterative methods, such as gradient descent and sparse linear solvers, emerge as natural candidates. While prior work [39] has demonstrated the value of this approach for gradient descent, applying it to sparse linear solvers requires investigating new techniques. We evaluate these adapted algorithms by modeling their data movement reductions against the corresponding impact on convergence rates. Finally, we conduct a speed-of-light analysis to project the maximum potential benefit of mapping these algorithms onto idealized PIM hardware.

4.1 Filter-Update Expressible Workloads

While modern workloads such as neural networks, graph analytics, and sparse scientific computation are heavily memory-bound, mapping them to Processing-In-

Memory (PIM) is not straightforward. As described in the previous chapters, in-memory logic often lacks the capability to perform complex arithmetic, such as double-precision floating-point operations or hardware multiplication [69]. Therefore, to successfully leverage the massive internal bandwidth of PIM [41], we must restrict the focus to algorithms that can be decomposed such that only simple, bandwidth-intensive operations are pushed to memory, while complex arithmetic remains on the host processor.

A workload is able to take advantage of the Filter-Update programming model if it possesses three key algorithmic characteristics. First, the workload must be memory bandwidth bound, where the cost of data movement strictly dominates over the cost of computation. Second, the algorithm must be iterative and tolerant to sparsification. Because the filter operation dynamically drops elements from the working set to save bandwidth, the underlying algorithm must be mathematically capable of converging even when computing partial or delayed updates (e.g., updating only the steepest dimensions of a gradient). Third, the selection criteria must be reducible to a simple element-wise predicate. The filter condition must be able to be evaluated for each element (such as a magnitude threshold) independently.

Many standard optimization algorithms naturally exhibit these traits. For problems that utilize iterative methods to converge to a minimum, the core computation often relies on memory-bound Sparse Matrix-Vector multiplication (SpMV) [70, 71]. To alleviate this bottleneck, prior work [72, 73] has explored software-level subspace splitting, where vectors are broken down and updated independently. While these approaches prove that iterative linear systems can converge without following the globally optimal descent direction in every step, software-only sparsification still requires transferring the entire working set across the memory bus just

to determine which elements to update. By expressing these algorithms into the Filter-Update model the sparsification step happens directly in memory and eliminates the data movement need. Even for stationary methods like Jacobi Iteration and Gauss-Seidel as introduced in Chapter 2 which do not naturally use filter and update operations, the properties of the algorithm enable this approach to still work. For these algorithms, there is no state that must be tracked between each iteration. The update can be computed on a per element basis and is tolerant to stale values (as can be seen from the difference in Jacobi Iteration and Gauss-Seidel). For this reason, selecting a subset of elements to update per iteration still allows the new effective iteration matrix to converge. While using less dimensions to compute the update per iteration hurts the rate of convergence, techniques outlined in Section 4.3 discuss techniques performed to ameliorate this problem.

While some iterative algorithms can be adapted for the filter update model, others are not amenable to using a subset to compute the update. The Biconjugate Gradient Stabilized method (BiCGSTAB) is a popular Krylov subspace solver for non-symmetric linear systems that relies on iteratively constructing mutually orthogonal (or bi-orthogonal) residual and search direction vectors to minimize the error over a specific subspace. When a filter is applied to these vectors, elements are dropped from the computation and the bi-orthogonality property of the vectors is lost. Once orthogonality is lost, the algorithm no longer spans the correct Krylov subspace, and the underlying mathematics completely break down, leading to rapid divergence or stalling.

4.2 Gradient Descent

Gradient descent is a first-order iterative optimization algorithm used to minimize a differentiable objective function. Given a function $f(x)$, with parameters $x \in \mathbb{R}^d$, the fundamental gradient descent iteration updates the parameters in the direction opposite of the gradient of the objective function $\nabla f(x_t)$ at the current parameter values x_t:

$$x_{t+1} = x_t - \eta_t \nabla f(x_t)$$

where $\eta_t > 0$ is the learning rate (step size) at iteration t.

In practice, computing the full gradient can be expensive, especially when the dataset is large. Stochastic gradient descent (SGD) addresses this by approximating the gradient using a single data sample or a small subset of the data (a "mini-batch"). Denoting the stochastic gradient at iteration t as g_t, the update rule becomes: $x_{t+1} = x_t - \eta_t g_t$. To accelerate optimization, SGD is often parallelized by distributing work across multiple nodes. The overall objective function is the average of objective functions computed on local data partitions at each of the N nodes:

$$f(x) = \frac{1}{N} \sum_{i=1}^{N} f_i(x)$$

At each iteration t, every node i computes a local stochastic gradient $g_{i,t}$ based on its local data and the current parameters x_t. Then, local gradients are aggregated across nodes to compute the update applied to x_t.

While distributed SGD can speed up computation, it introduces significant communication overhead, since nodes need to exchange local gradients at each iteration. For functions with millions or billions of parameters, this communication

may become a performance bottleneck [74][75].

4.2.1 Data Movement Reduction Analysis

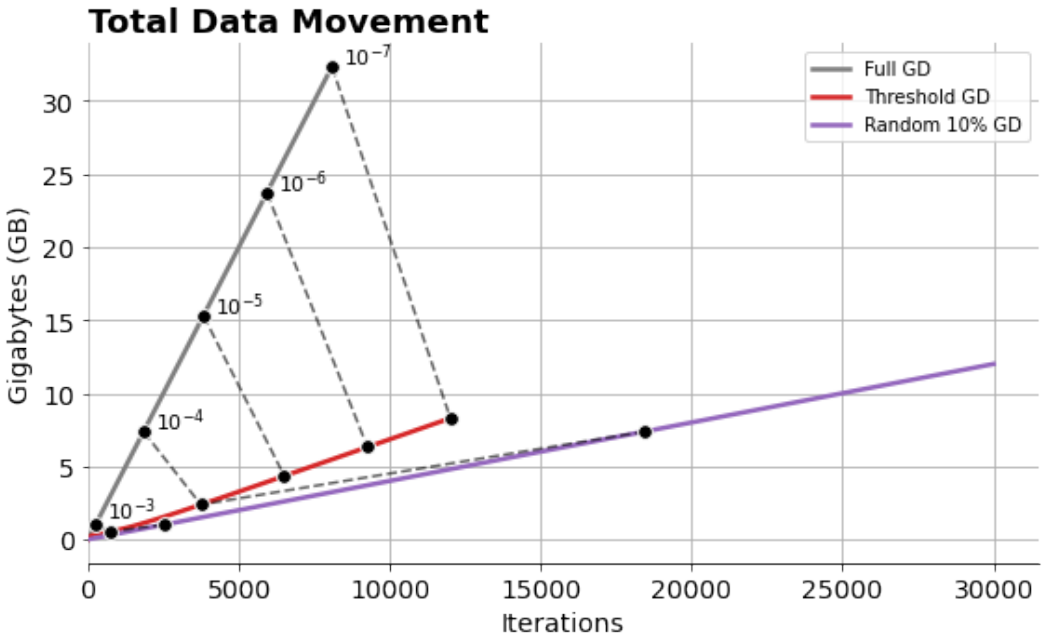

Figure 4.1: Cumulative data movement until convergence over iterations for different gradient descent methods with varying sparsification strategies. The points on each line mark the iteration where the residual falls below 10^{-j} for j from 3 to 7. The threshold-based filtering approach requires least data movement for each isoline with as much as 3.9x less when compared to full gradient descent. The random filtering approach requires about the same amount of data movement as Full GD but 10x more iterations.

To validate this approach, we investigate gradient descent and utilize various filters to reduce the amount of data movement required. For each of these filters, we observe the number of iterations required before convergence along with the amount of data transferred per iteration along with the total amount of data transferred before convergence. A random quadratic objective function of the form:

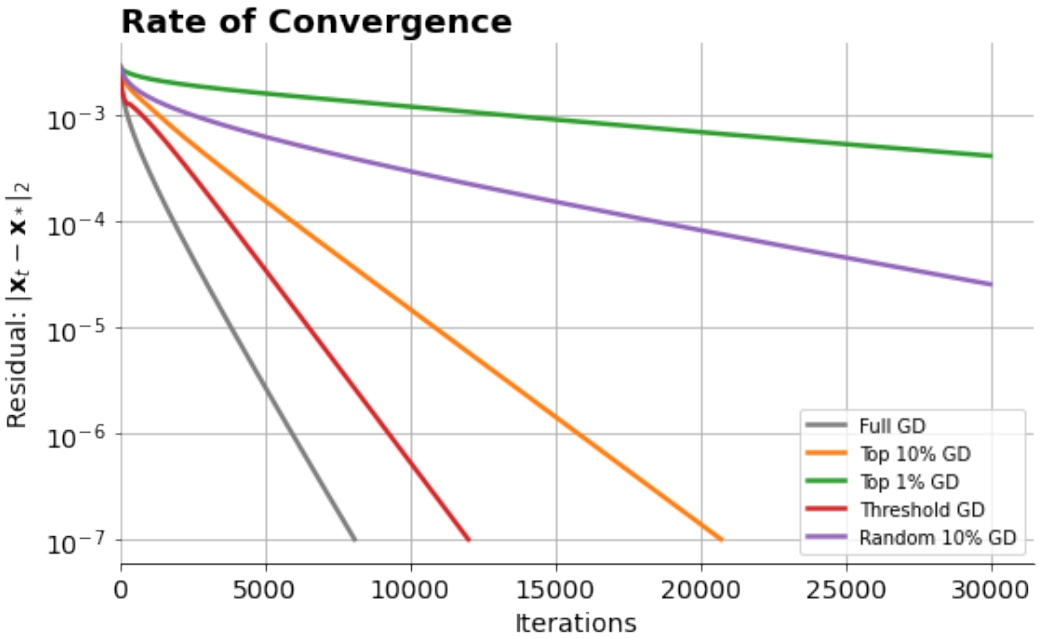

Figure 4.2: Convergence behavior of gradient descent under different gradient sparsification strategies on an ill-conditioned quadratic objective function. The residual computes the error between the value of x_t in iteration t and the known minima x_*. x_t is considered to have converged when the residual $\leq 10^{-7}$. The filter approaches both converge at a rate comparable to the full gradient descent while requiring less data movement.

$f(x) = \frac{1}{2}\sum_{i=1}^{n} c_i x_i^2$ where $x \in \mathbb{R}^n$ is the optimization variable. c is initialized using values distributed across a logarithmic scale to create an ill-conditioned problem. This helps simulate a common problem in optimization where some dimensions have a steeper curvature than others and thus produce larger gradients from small perturbations of x. This approach serves to highlight the effects of using a filter to sparsify the problem. The threshold filter selects the elements of x where the magnitude of the gradient is above a certain value. This filter easily maps to the PIM hardware as each element can be evaluated independently. The top k filter selects the k elements of x with the largest gradient magnitude. While the data dependency in this filter does not map easily to PIM hardware, it serves as a comparison to the threshold filter. The third filter that is used is a random k filter where a random set of k elements are selected to be updated during each iteration.

To quantify the benefits of employing a magnitude-based filtering and sparse updates executed in PIM, we compare the DRAM-CPU data traffic against a traditional dense CPU-based update scheme.

We define the following parameters for our data movement analysis: D denotes the length of the parameter/gradient vector in DRAM, β represents bytes per numeric value (e.g. 4 bytes for FP32), and α indicates bytes per index in sparse representation (4 bytes for int32). The magnitude filtering threshold is τ, while p_w and p_g represent the pass ratios (as fractions of D) for the weight update vector and gradient returned to PIM, respectively. We use x_t for the parameter vector at iteration t and g_t for the general gradient at iteration t, with learning rate η.

For conventional dense implementations, we consider two scenarios. In the *best case*, the gradient g_t is produced and applied while cached in the processor, requiring only parameter reads and writes which gives: $B_{\text{dense(best)}} = 2\beta D$. In the

worst case, the gradient spills to DRAM before the update, requiring additional gradient storage and retrieval which leads to: $B_{\text{dense(worst)}} = 4\beta D$.

Our PIM-based approach filters the dense gradient using the magnitude threshold τ, with the CPU performing scaling operations on sparse values before PIM scatter-adds them back. We analyze two strategies for transferring data between the CPU and PIM. When both *indices and values* are transmitted bidirectionally, the data movement requirement is $B_{\text{PIM(idx+val)}} = 2(\alpha + \beta)(p_w + p_g)D$. Alternatively, for some applications it is possible to use an *index-only outbound* strategy, where only indices are sent from DRAM to CPU, which results in: $B_{\text{PIM(idx-only)}} = \alpha p_g D + \beta(p_g + p_w)D$.

The data movement reduction that can be achieved for the indices and values scenario and the index-only scenario is:

$$\text{Data Transfer Reduction}_{\text{PIM(idx+val)}} = \frac{B_{\text{dense}}}{2(\alpha + \beta)(p_w + p_g)D}$$

$$\text{Data Transfer Reduction}_{\text{PIM(idx-only)}} = \frac{B_{\text{dense}}}{\alpha p_g D + \beta(p_g + p_w)D}$$

Assuming $\alpha = \beta$ (32-bit indices and values) and setting $p_w = p_g = p$ for simplicity, we obtain the bandwidth reductions shown in Table 4.1.

Table 4.1: Bandwidth reduction factors for different sparsity levels

Sparsity Level	p	**Idx + Val**	**Idx Only**
50% sparsity	0.50	1–2×	1.3–2.7×
90% sparsity	0.10	5–10×	6.7–13.3×
99% sparsity	0.01	50–100×	67–133×

The analysis demonstrates that with 90% sparsity ($p = 0.1$), our PIM pipeline reduces external traffic by 5–10× for the indices + values approach and up to 13×

for the index-only approach. These gains increase significantly as sparsity deepens, making the approach particularly attractive for applications with highly sparse gradient updates.

4.2.2 Filtering Approaches

We explore three filtering strategies to investigate the tradeoffs between effectiveness and hardware feasibility. The top 10% filter selects all components of the gradient that correspond to the top 10% of gradient magnitudes. Although this filter cannot be easily implemented in PIM, it serves to highlight the potential effectiveness of always selectively updating the most significant components. The hard threshold filter, which can be easily implemented in PIM, selects all values of the gradient that are above a certain threshold. This hard threshold is determined by investigating the initial values of the gradient vector and setting a threshold that would select about 10% of the data. The hard threshold is then lowered by 1% after each iteration where no gradient values are above the threshold. The random 10% filter serves as a baseline by selecting a random 10% subset of indices in each iteration, independent of gradient magnitude. The results for each of these filters can be seen in Figure 4.2 and Figure 4.1. The results for the random 10% filter are missing in Table 4.2 as the approach was not able to converge within a reasonable amount of time on the machine used to run this model.

Table 4.2: Convergence and Data Movement for Different Filtering Techniques

Method	**Iterations to Converge**	**Updates per Iteration**	**Total Updates (GB)**
Full	8,083	1,000,000	32.3
Top 10%	20,720	100,000	8.28
Hard Threshold	12,007	172,624	8.29
Random 10%	—	100,000	—

4.2.3 Potential Performance Gain

The hard threshold filtered gradient descent approach converges to the same tolerance as full gradient descent while transferring significantly less data. As seen in Table 4.2, in order to converge to 10^{-7}, machine epsilon for 32-bit floats, the full gradient descent approach requires about 8,000 iterations and transfers 32 GB of data across the memory bus. For a hard threshold gradient descent approach, only 8 GB of data is transferred while the number of iterations increases by 50%. Using a random filter which avoids using PIM, the residual never converges. For the lower tolerances where the random filter converges, the same amount of data as the full gradient descent is transferred while requiring 10x more iterations.

In Fig. 4.2, the rate of convergence for each of these approaches can be seen. The fastest rate of convergence occurs with the traditional gradient descent approach that utilizes the entire dataset as expected. In contrast, the hard thresholding filter utilizes about 17% of the data as the traditional method per iteration while the number of iterations required to converge increases by 1.5x. When compared with a hypothetical top 10% filter the number of iterations required to converge is 2.5x higher while incurring 10x less data movement per iteration. Due to the difficulty of implementing the Top k filter in hardware and the slower rate of convergence when compared to hard thresholding, this filtering technique does not fit well for filter/update computation in memory. When exploring more aggressive filters such as the top 1%, we find that additional error feedback computation [40] is needed to converge. For random 10% filter which avoids using compute in memory, the residual decreases at a far slower rate and does not converge to below the machine epsilon for 32 bits floats (10^{-7}) when allowed 5x more iterations than the baseline. Fig. 4.1 shows the total data movement for the feasible filter approaches along with

the iso-convergence lines across these methods. The hard threshold filter utilizes the least data movement for each isoline and is as much as 3.9x lower than the traditional gradient descent approach for 10^{-7}. The random 10% filter requires about the same amount of data movement as the full gradient descent implementation to converge to each isoline. However, since this filter moves 10x less data, this requires 10x more iterations and is thus infeasible for converging to lower precision values.

For these applications with sufficient parallelism and low arithmetic intensity, data movement time can be used as a first-order model to approximate performance. In this model, performance is estimated as the sum of the CPU compute time, data movement time, and PIM compute time if applicable. Here we assume the problem is large enough to saturate both the DRAM bandwidth and PIM internal bandwidth. From this we get the following equation: $T = I(\frac{mF}{P} + \frac{m}{B} + \frac{D}{B_{PIM}})$ where I is the number of iterations until convergence, m represents the total bytes transferred per iteration, B is the bandwidth (bytes/sec) of the system, F is the number of floating-point operations required on the CPU for each byte for a given objective function, P is the number of floating point operations per second the CPU is capable of, D is the size of the vector, and B_{PIM} is the internal bandwidth for PIM inside DRAM. This model assumes that for the filter operation the PIM computation on each bank can overlap with the data movement time of the filtered result that is processed on the host side. This is due to the fact that the PIM compute unit in each bank processes an independent set of data, allowing each unit to work in parallel. For update operations, since the updated values are written to DRAM, the PIM compute unit of each bank is able to independently fetch and write back the updated values. The internal bandwidth for PIM can be approximated with $B_{PIM} = BXG$ where B is the external DRAM bandwidth, X is the bank-level parallelism, and G is the global

data line.

We estimate the performance for a system with an Intel Xeon E5-2640v4 with DDR4 memory using the results from the full gradient descent and filtered gradient descent. These results show hard threshold gradient descent could lead to a 3.9x speedup over the full gradient descent. In this system, the data movement time dominates the total runtime of the traditional gradient descent approach. For in memory computation, the PIM latency does not significantly change application runtime since the internal PIM bandwidth that can be utilized to perform filtering and update operations on the entire dataset is 128 times faster than the external DRAM bandwidth. This thereby enables the hard threshold gradient descent approach, which reduces the total data movement by 3.9x, to be 3.9x faster than the baseline since the PIM compute time does not significantly affect runtime.

For filtering approaches like Top 10% which are unable to use PIM hardware, the whole dataset must be transferred each iteration while the amount of computation is reduced. In this case, the reduced amount of computation leads to a slow down since this approach will require more iterations. For memory-bound iterative methods which can be cast into filter/update computational patterns, the sparsification of each iteration must be balanced with the increased number of iterations that will be required in order to achieve good performance.

4.3 Filter-Update Modified Linear System Solvers

For these linear system solvers, the key insight is derived from the intuition that from a random initial condition, some dimensions will naturally have a larger error than others while other dimensions may already sit very close to the true solution. For this reason, it makes sense to focus the computation upon the dimensions that

are more incorrect in order to compute an update that is generally in the correct direction. The method of determining which dimensions are more or less correct is determined by selecting the elements of the residual vector, $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$, which have the largest magnitude.

4.3.1 Types of Filtering Approaches

Threshold. For the threshold filter, all elements where the magnitude is greater than the specified value are passed through. This threshold is determined using the residual values from the initial starting residual. Following this, after each iteration the threshold is lowered by a fixed constant. If no values were passed through the filter in the last iteration the threshold value is halved. The threshold has a floor value that is determined by the machine ϵ.

Selecting an appropriate decay rate is essential for having an effective filter operation. For thresholds that decay too quickly the filter operation allows most dimensions to pass through which thereby equates to having no filter. When the elements per iteration that are selected are simply the first n elements that were seen with no significance granted towards the the error for each dimension, the number of iterations required to converge is increased by a factor that results in transferring the matrix back and forth from main memory the same number of times as the baseline. For instance, when filtering for 10% of the dimensions in each iteration, the 10x more iterations are required to converge when compared to performing no filter operations. When the threshold value does not decay fast enough, the number of dimensions that pass through the filter ends up being less than the maximum allowable number of elements to be transferred in each iteration. This results in the update computation using less information than could be used and thus also leads

to a less accurate update computation and more iterations required to converge. For this reason, the decay rate of the threshold value is determined by a formula that is based upon 3 variables - the spectral radius of the iteration matrix, the percentage of the matrix used in each iteration and the number of colors the matrix has. These variables are put together to form the following formula $\rho^{Cp^2/2}$. Here the percentage of the dimensions being filtered and the number of colors (a variable which will be discussed in Section 4.3.2) are important as they inform the number of iterations that are required before the entire matrix will be seen. ρ remains as a property of the matrix itself and can be computed as described earlier in Section 2.3.1.

Top k%. This approach serves as an ablation study for the threshold filter. For a threshold that is set perfectly in each iteration, the results that are passed through the filter should match the output from the "top k%" filter. While this filtering approach presents the perfect thresholding approach, it remains difficult to implement practically in PIM hardware. This is due to the fact that in order to determine the top k% the entire vector must be sorted which thus makes it difficult to determine whether an individual element should be included in the filter. Now the required state to determine whether an element belongs in the output set is the same size as the entire vector. For this reason, this filtering approach is not used when evaluating runtime and is only used to assess the number of iterations required to converge.

When comparing the number of iterations required to converge between the threshold and top k filters, we can see the effectiveness of the threshold lowering formula. Here, the focus is on studying these two matrices, where for larger sizes the matrix is replicated in a block diagonal fashion with an addition 20% of data added randomly across the matrix to ensure that the spectrum of the larger matrix

Table 4.3: Iteration counts to convergence for Filter-Update Gauss-Seidel, comparing threshold-based and top-k filter selection across scaled `bcsstk08` and `bcsstk09` systems.

					Iterations	
Matrix	**Size**	ρ_J	**Dimension**	**Nonzeros**	**Threshold**	**Top-k**
`bcsstk08`	×100	1.84	107,400	1,349,064	43,749	37,548
	×1000		1,074,000	13,495,464	43,551	37,269
	×2000		2,148,000	26,991,464	42,948	36,927
	×3000		3,222,000	40,487,464	41,976	36,927
	×5000		5,370,000	67,479,464	41,949	36,936
	×10000		10,740,000	134,959,464	43,587	36,990
`bcsstk09`	×100	1.93	108,300	1,897,160	157,660	132,120
	×1000		1,083,000	18,976,460	158,560	127,280
	×2000		2,166,000	37,953,460	365,030	125,420
	×3000		3,249,000	56,930,460	315,850	124,800
	×5000		5,415,000	94,884,460	344,500	124,370
	×10000		10,830,000	189,769,460	142,240	124,160

is not identical to the smaller sizes. While it can be seen that the Top k approach always requires the least number of iterations to converge, using this formula to set the threshold keeps the number of iterations with 13.6% - 17.8% for the bcsstk08 matrix. Here, these matrices represent real structural problems [1] from Matix Market [76] rather than synthetic benchmarks.

4.3.2 Coloring the Matrix

In order to ensure convergence, a graph coloring algorithm on the matrix must be performed. Since the A matrix is often sparse for real world applications, we can compute an update for elements that have no effect on each other. By doing this, the update that is computed in each iteration for a batch of elements can be parallelized since these dimensions do not effect one another. More formally, two dimensions i and j can be in the same color if the $a_{ij} = 0$ and $a_{ji} = 0$. Following this coloring

operation, the filter is applied to a single matrix color during each iteration. Then in each iteration, a different color is filtered.

When this coloring step is omitted, simultaneously filtering and updating dependent dimensions violates the strict sequential dependencies inherent to methods like Gauss-Seidel. If two dependent elements are updated in parallel, the computation effectively uses stale values from the previous iteration rather than incorporating the most recently computed updates. This degrades the spectral properties of the iteration matrix. The filter-update modified algorithm then converges far more slowly or, for many matrices, fail to converge entirely. While computing the graph coloring introduces an upfront preprocessing cost, this overhead is heavily amortized over the lifespan of the solver. Because these iterative algorithms typically require thousands of iterations to reach a sufficiently small residual, the performance gained by enabling safe, massively parallel PIM updates in every subsequent iteration vastly outweighs the one-time setup time.

The coloring algorithm used to separate the bank has a few key properties. This algorithm prioritizes sorting each dimension into colors in a fashion that minimizes the variance in the size of each color. For results where certain colors are significantly larger than others, more total iterations are required in order for all elements to be be seen by a specific filter. For instance if one color contains twice as many elements as another and each iteration selects 10% of the elements within a specific color, the worst case scenario will involve 20 iterations to select every element from the larger color versus 10 iterations in the smaller color. As the filter-update algorithm uses a round robin scheduling to cycles through each color, the color with fewer elements will have a larger focus than the larger color.

4.3.3 Relaxation Constant

In order to accelerate convergence, a constant ω is used to damp the update computed in each iteration. This constant, as introduced in Section 2.3.3, serves as a relaxation factor that can significantly improve the convergence rate of slowly converging iterative processes.

In classical Successive Over-Relaxation (SOR), the optimal relaxation factor (ω_{opt}) can often be calculated analytically based on the spectral radius of the Jacobi iteration matrix. However, this approach reflects the iteration matrix for when the entire matrix is used to compute the update in each iteration. The Filter-Update modified approach lacks the same convergence behavior as a fraction of the dimensions are used in each iteration. Because the algorithm applies a magnitude-based threshold to the residual, the subset of dimensions updated in any given iteration is highly dynamic and data-dependent. Consequently, the effective iteration matrix is not static, which leads to analytical formulas for ω_{opt} to be inaccurate for predicting the optimal relaxation factor under heavy sparsification.

Given the dynamic nature of these filtered sweeps, the relaxation constant in this work is determined empirically. Profiling the solver across a small subset of initial iterations allows for heuristic tuning of ω. Future work may explore adaptive relaxation techniques, where ω is dynamically adjusted at runtime based on the variance in the sizes of the matrix colors and the current threshold decay rate.

4.3.4 Summary

When putting all of these approaches together, the resulting computational flow per iteration can be seen at a high level in Listing 4.1. Before the iteration begins, the coloring algorithm separates the dimensions of the A matrix such that updates

computed within the same sweep do not collide with one another. The rows and ordering of the elements in the A matrix and in the solution vector are then reordered appropriately such that each color lies in a contiguous set of rows. Following this, in each iteration, a single color is selected to be filtered and updated. The color that is selected is chosen in a round robin fashion such that all colors are seen once before a color is seen for a second iteration. In the filter phase, the first n elements that pass the magnitude based threshold filter are returned where n represents the maximum updates per batch. This should be sized such that the active set of rows to be updated are able to fit within cache on the host side. As the compute phase uses the solution vector, $\mathbf{x}$ and not the residual vector, $\mathbf{r}$, only the indices that pass the filter are returned and the corresponding residual value at each index is not needed. To prevent starvation within a color, the starting search index for the filter is dynamically updated to begin exactly where the previous batch finished. This state is managed by the coloring.range(c) argument passed to the filter. In the compute phase, the active indices that are returned from the filter are used to compute the updates for those values of $\mathbf{x}$. The updates for $\mathbf{x}$ and $\mathbf{r}$ which represents the residual are accumulated in near memory hardware. The threshold is then decayed based on the constant discussed earlier. Following this, the convergence criterion is checked using the updates to determine the change in the residual.

4.4 Filter Update Solver Performance Evaluation

The performance of using filter and update operations is done using a performance model of the filter and update operations in near-memory hardware. While these operations have been mapped to commercial PIM systems like UPMEM, the inability for these systems to achieve the full DDR4 bandwidth of the interface that

```
for (int sweep = 0; sweep < max_sweeps; sweep++) {
    const Color c = coloring.color_for(sweep);   // round-robin

    // --- Filter phase ---
    ActiveSet active = pimFilter(r_pim, ABS_GREATER_THAN, threshold,
                                 coloring.range(c),
    max_updates_per_batch);

    if (active.empty()) {
        threshold.relax(); // half the threshold if nothing passed
        continue;
    }

    // --- Compute phase ---
    // Reads A and x only, and only the rows named by the active set.
    Corrections d = computeUpdates(A, x, omega, active);

    // --- Update phase ---
    pimUpdate(x_pim, d.x, SUB);
    pimUpdate(r_pim, d.r, SUB);

    threshold.decay();
    if (check_convergence(x, d.x, tol)) break;
}
```

Listing 4.1: The filter-update formulation of Gauss-Seidel. Each sweep filters the residual for the rows of one color whose magnitude exceeds the current threshold, computes the corresponding corrections on the host, and scatters them back into the solution and residual vectors. No data movement is expressed by the programmer.

these hardware modules reside in make it difficult to compare these results to those of traditional state of the art systems. By diminishing the memory bandwidth of the system, it becomes difficult to accelerate these iterative algorithms which are already memory bandwidth bottlenecked. For this reason a performance model of a PIM system that achieves the full traditional bandwidth is used to evaluate these machines. As ongoing work as described in Section 3.3, future PIM hardware may be able to sustain full DRAM bandwidth while performing filter and update operations.

4.4.1 Speed-of-Light Performance Model

As a speed-of-light evaluation, the performance of these algorithms is measured by only measuring the data movement time of the filter and update operations. This allows for the existing results to separate the performance effects of the limitations of compute near memory from the effects of reducing the amount of data movement to alleviate the memory bandwidth bottleneck. This model for performance, while obviously faster, should remain close to what the potential peak performance of these operations could be due to the simplicity of the computation required. Many of the filter and update operations involve a single accumulation or comparison operation to be performed with a constant value or a single other operand. These operations are element-wise and do not require any communication with other data that may be resident in other regions of memory. This measurement is done by performing the filter and update computation during an untimed portion of the computation and copying the result to a buffer. Following this, the time to perform a memcpy operation of the results is measured. The computation to be done in each iteration is then performed using the data from the buffer that has been copied and the results are written back to the appropriate buffers. The time for this computation is timed and thus resembles the authentic performance on modern CPUs for the type of computation and memory access pattern of these algorithms.

4.4.2 Peak Filter-Update Performance Evaluation

For sparse linear systems, memory bandwidth is the bottleneck for solvers like Gauss-Seidel. By adapting this numerical method using filter and update operations, an important subset of the linear system can be selected in each iteration and thereby reduce the amount of data movement required. While the more iterations

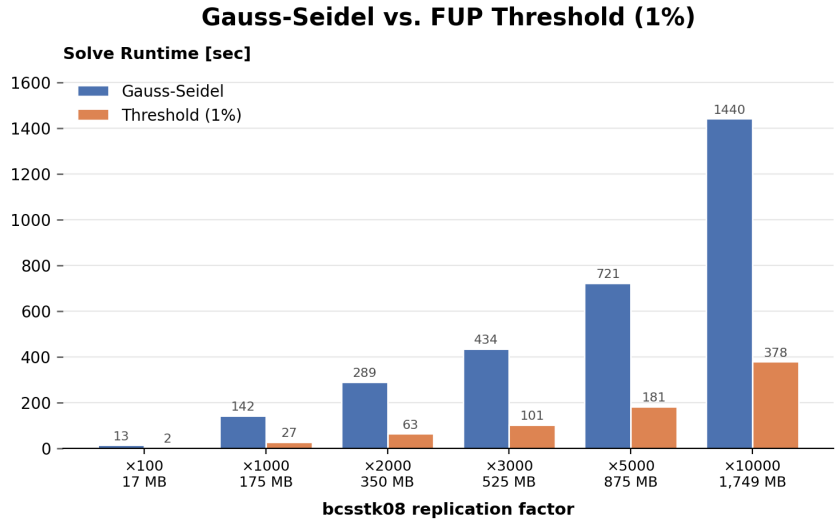

Figure 4.3: Runtime for solving a linear system using Gauss-Seidel versus a speed of light model for Filter-Update modified Gauss-Seidel. Both algorithms are run until convergence.

are required to converge, the cheaper cost to compute the update in each iteration is able to offset the increase in number of iterations. This difference can be seen in Figure 4.3. The difference in number of iterations required to converge can be seen in Table 4.4. In this case, the bcsstk08 [1] matrix from Matrix Market [76] is used and the initial x_0 is the 0 vector. Convergence in this case is defined as $\frac{||r||_2}{||b||_2} < \epsilon$ where $r = b - Ax, \epsilon = 10^{-6}$.

The filter operation used is a threshold operation $|r_i| < threshold$ where the threshold is initially set before beginning convergence by computing the initial residual.

These numbers were measured on a system with a 16 core Intel i7-10700 which has a 16MB L3 cache size. This system also has 64 GB of DDR4-2600 DRAM which provides a theoretical peak bandwidth of 38 GB/s. Using the STREAM benchmark suite [77], the copy peak is 34.6 GB/s.

Table 4.4: Baseline Gauss-Seidel versus the Filter-Update threshold solver on replicated `bcsstk08` [1] systems. Runtimes are wall-clock seconds to a relative residual below 10^{-6}.

			Gauss–Seidel		Filter–Update		
Replicas	n	nnz	Time (s)	Iters	Time (s)	Iters	Speedup
100	107,400	1,349,064	12.84	10,742	2.04	43,749	6.29×
1000	1,074,000	13,495,464	141.93	10,367	27.28	43,551	5.20×
2000	2,148,000	26,991,464	289.41	10,287	62.90	42,948	4.60×
3000	3,222,000	40,487,464	433.52	10,269	97.17	41,976	4.46×
5000	5,370,000	67,479,464	720.86	10,242	184.41	41,949	3.91×
10000	10,740,000	134,959,464	1440.41	10,224	378.08	43,587	3.81×

For the Gauss-Seidel baseline, the algorithm is parallelized across all cores on the machine. Here the update computation to be done in each iteration is parallelized across cores where each core computes the update for a different element of the vector. When evaluating the performance of this implementation, the effective throughput is measured to be 27.9 - 32.8 GB/s. This confirms that the bottleneck of this algorithm arises from streaming the matrix back and forth from DRAM in order to compute the update in each iteration. However, this data movement requirement can be alleviated by focusing on significant subset of rows to update in each iteration.

Table 4.5: Peak machine DRAM performance

Test	Throughput (GB/s)
copy	34.632
scale	24.195
add	26.939
triad	27.109

In the filter-update modified approach, 1% of the dimensions are used in each iteration which thereby reduces the data movement requirement in each iteration. The dimensions to be updated in each iteration are selected by selecting the elements

of the vector where the magnitude of the element in the residual vector is largest. By computing the update for these elements in the output vector, the update that is computed in each iteration is able to focus upon the dimensions where the error remains largest and ignore computation upon elements which are already close to the correct value. In addition, this smaller set of elements allows this portion solution vector to remain resident within the cache on the host side and reduce random memory accesses to DRAM.

For linear systems where the dimension ranges from around 100,000 - 10,000,000, the filter-update Gauss-Seidel approach ranges from 3.81 - 6.29x faster to converge. While 100x less data is used in each iteration, the 4.07 - 4.26x increase in number of iterations when using the filter-update Gauss-Seidel approach offsets much of the performance gain from doing less computation per iteration. This performance model can be used to estimate the required computational throughput and internal bandwidth that is required. When looking at the largest matrix investigated, the speed of light estimate of the filter update runtime means that $T_{PIM} = T_{GS} - T_{SOL} = 1145.6$ seconds. As the matrix is divided into 9 colors, filtering a single color requires reading 10,740,000 / 9 elements = 1,193,333 elements which is 9.55 MB. As each update consists of 1% of the data for both the solution and residual vectors, 3.44 MB which leads to a total of 12.99 MB per iteration. Across the total number of iterations this leads to 566.2 GB. A similar computation can be used to estimate the number of arithmetic operations. A filter operation requires a single floating point operation per element while the update operation also requires a single floating point operation per element. In total, this means 1.41 MFLOPs are required per iteration or 61.46 GFLOPs until convergence. By serializing these operations in memory the minimum performance requirements of the

system to match the baseline can be found. From this, an internal bandwidth of 988 MB/s and a compute throughput of 107.3 MFLOPs are the minimum requirements for matching the CPU baseline if the memory bandwidth of the PIM-enabled memory matched DDR4-2600 speeds on this system. As both of these numbers are well within the peak capabilities of modern PIM architectures, it stands to reason that even software emulated floating point on basic RISC cores can clear this threshold.

An interesting trend can be noticed over the runtime of each iteration of the filter-update modified solver. As the solution vector approaches a residual of 0 the computation in each stage becomes more expensive. This is due to a property of these stationary iterative methods, that is actually exploited in many practical applications. Stationary iterative methods like Gauss-Seidel are not usually run to convergence. Instead, they are used within hierarchical frameworks such as Multigrid methods where Gauss-Seidel is used as a smoother or relaxation operator [72, 78, 79]. These methods are slow when eliminating global, low frequency errors but are very effective for damping high frequency errors within just a handful of iterations. As a result, multigrid solvers typically execute Gauss-Seidel for a few iterations before restricting the residual to a coarser grid. The effectiveness of these methods for high frequency error further explains the effectiveness of the filter update modified approach for the earlier iterations than when compared to the later ones. When beginning from a random point and in the early iterations, the error is chaotic and elements fluctuate wildly in sign and magnitude. When looking at the true error $\mathbf{e} = \mathbf{x}_{solution} - \mathbf{x}$, the residual is $\mathbf{r} = \mathbf{Ae}$ or per element this gives $r_i = \sum a_{ij} e_j$. This means, when filtering the residual vector, the filter selects a wide, random assortment of rows. However, after a few iterations of Gauss-Seidel, the high frequency errors in the solution are dampened which results in elements of the error,

e, to have much more similar sign and magnitude. This now means the terms in the summation for each element of the residual do not cancel out from randomness in the error vector and now higher-degree nodes which sum many small error terms have a higher residual. For this reason, in the later iterations, the filter operation will naturally select rows with a higher average degree and will require longer to compute the update. While putting together the full end-to-end application is discussed in Chapter 6 as a potential direction for future work, here we investigate the performance of the Filter-Update modified solver for a fixed number of iterations to mimic how these stationary methods are often used in larger applications.

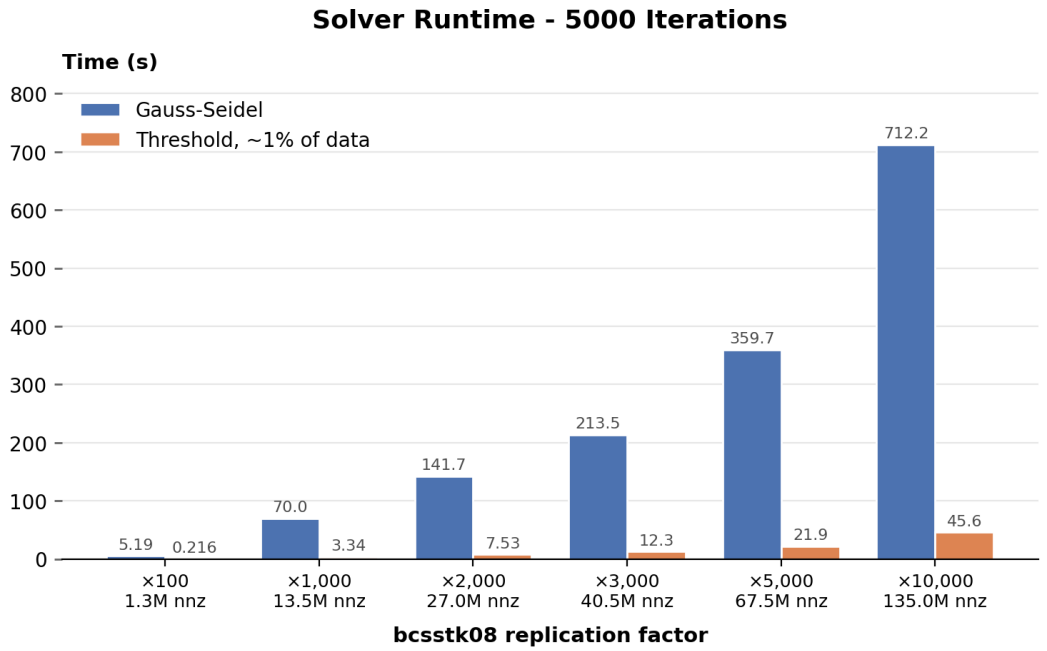

Figure 4.4: Runtime for 5,000 iterations of Gauss-Seidel vs. Filter-Update modified Gauss-Seidel across synthetic matrices of various sizes. The runtime when using a threshold-based filter that is set such that about 1% of the dimensions are used to compute the update in each iteration is as much as 24.1x faster based on a speed-of-light performance model for the filter and update operations.

Focusing on a set number of iterations , the speedup from using the filter-

update approach is far more dramatic. When selecting a subset of the matrix that is 1% of the entire system, a 15.6 - 24.1x speedup can be seen from using the filter-update Gauss-Seidel solver over Gauss-Seidel for 5000 iterations as shown in Figure 4.4. Here smaller sizes tend to have larger speedups while the larger sizes show slightly less. This effect becomes even more extreme as the number of iterations is lowered even further to 1000 iterations as seen in Figure 4.5. Here the speedup across the same matrices range from 27.4 - 40.2x. These results further demonstrate the trend that the computation phase becomes more expensive as the filter focuses on nodes of higher and higher degree after more iterations.

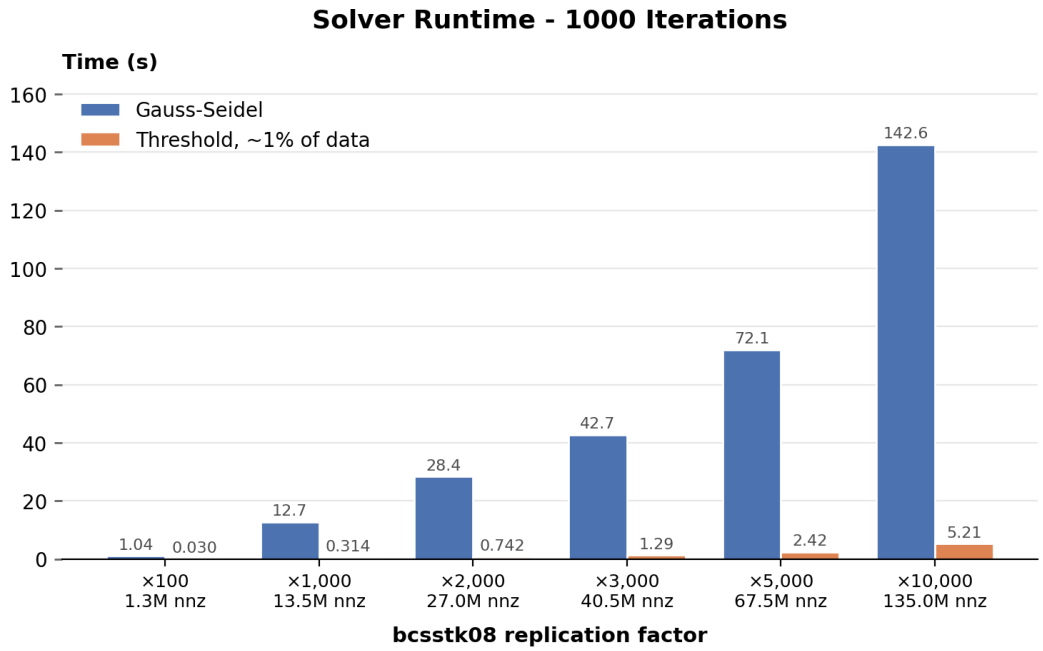

Figure 4.5: Runtime for 1,000 iterations of Gauss-Seidel vs. Filter-Update modified Gauss-Seidel across synthetic matrices of various sizes. The runtime when using a threshold-based filter that is set such that about 1% of the dimensions are used to compute the update in each iteration is as much as 40.2x faster based on a speed-of-light performance model for the filter and update operations.

Here the result from running a set number of iterations can be expected. By

fixing the number of iterations, the filter-update solver performs less compute and arrives at a less accurate solution than the standard Gauss-Seidel output. Instead, another approach is to set a looser convergence threshold. As these fixed-point iterative methods are not typically run to the true solution, a much earlier stopping point can be used instead to focus on damping high frequency noise. Here, the convergence criterion is set by running until the relative residual is less than 0.5 as seen in the following equation.

$$\|\mathbf{r}^{(k)}\|_2 < \tau\|\mathbf{b}\|_2, \qquad \mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}, \qquad \tau = 0.5. \tag{4.1}$$

The results from this experiment can be seen in Figure 4.6. Initializing with a zero vector, the number of iterations required to converge for Gauss-Seidel ranged from 880-951 while the number of iterations for the filter-update model was 3400-3700 iterations. As a result, the filter-update model achieves a 4.2 - 7.0× speedup over Gauss-Seidel across the evaluated sizes for this earlier stopping condition. Here, the speedup is still higher than when running until the traditional convergence criterion but the 3.6-4.5× increase in number of iterations for the filter-update model does dampen the possible performance gain. As discussed in Chapter 6, determining the optimal filter-update stopping criterion for integration into broader Multigrid methods remains an important area for future work.

Given these results from a speed-of-light analysis, the benefit from adapting these iterative numerical methods to the filter-update model can be seen. When starting from a poor initial guess, the filtering operation enables computing upon a smaller portion of the data while providing an update that is still generally in the correct direction to enable convergence. From this exploration, it can be seen that the increase in number of iterations required to converge is still small enough such

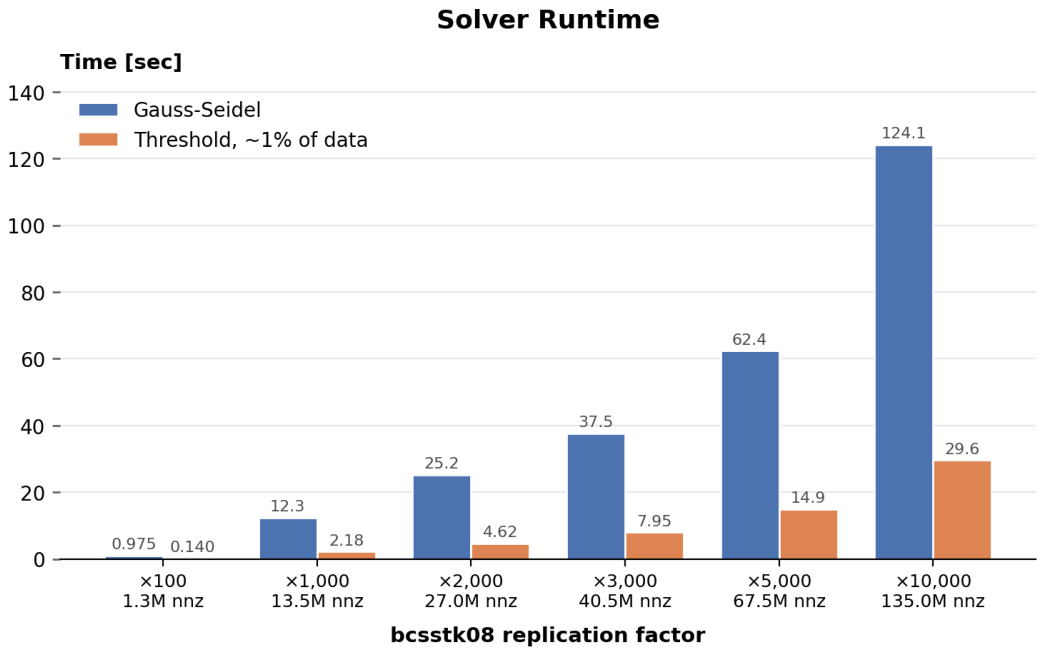

Figure 4.6: Runtime for Gauss-Seidel and Filter-Update modified Gauss-Seidel while using the $\|\mathbf{r}^{(k)}\|_2/\|\mathbf{b}\|_2 < 0.5$ stopping condition. The threshold is set such that approximately 1% of the data is passed through the filter in each iteration. Under this earlier stopping condition, the filter-update approach achieves a speedup of up to 7×, slightly exceeding the performance gains observed under traditional convergence criterion.

that this problem sees as much as a 6.8x speedup over the traditional approach. As this initial guess is refined to be closer to solution, other state-of-the-art methods may be better fit to get an exact answer. Given these initial results, further research on the algorithm side remains to explore how these filter and update operations can be leveraged in other applications to make use of future PIM systems.

4.5 Summary

This chapter demonstrated how various iterative algorithms can be effectively expressed using the Filter-Update model. For optimization algorithms like gradient descent, the filter-update pattern is a natural fit that has been explored in prior literature, making its mapping to this programming model straightforward. However, for other domains, such as sparse linear solvers, this chapter illustrated how algorithms must be explicitly redesigned to take advantage of near-memory acceleration on systems with highly constrained computational capabilities. Ultimately, as established by the Filter-Update model, utilizing a declarative specification over regions of memory enables effective hardware-software co-design, allowing novel algorithms to successfully leverage near-memory compute.

However, memory-centric systems encompass a much broader design space than capability-limited PIM hardware. Emerging and future systems integrate more general-purpose logic and specialized accelerators within a shared address space. This can be seen from the emergence of CXL Type-2 devices [37], buffer-integrated FPGAs like Samsung AxDIMM [50], and logic-integrated memory like Aquabolt-XL [80]. For these highly capable platforms, restricting computation to simple filter and update operations leaves significant hardware potential underutilized.

To fully exploit these advanced architectures, the following chapter intro-

duces the Magic Memory programming model, which provides a more generalized abstraction. When the underlying hardware is capable of arbitrary, complex computation, the declarative specification can be elevated from simple element-wise operations to persistent mathematical invariants. Chapter 5 explores how declaring these invariants between memory regions allows shared-memory systems to autonomously manage complex sparse data formats and dynamically orchestrate how and when computation occurs.

Chapter 5

Magic Memory

As established in the thesis of this dissertation, the extent to which data orchestration can be abstracted away from the programmer depends on the capabilities of the underlying hardware. While previous chapters addressed limited-compute PIM architectures, this chapter explores memory-centric systems equipped with more general-purpose logic and a shared address space. For these systems, we introduce the Magic Memory programming model. In this approach, programmers provide a declarative specification over memory regions by defining computations as persistent invariants. This chapter demonstrates that this specification carries sufficient information for the system to manage data orchestration, allowing it to simplify managing complex sparse data structures and perform incremental computation behind the scenes. Finally, the feasibility of this model is evaluated through a prototype CPU-FPGA implementation of an interactive PageRank application.

5.1 Magic Memory Programming Model

For memory-centric systems where more general purpose computation is available near memory, a programming model that exploits that capability is needed. With a wider range of capabilities, the declarative specification between regions of memory that describes computation can be generalized to a mathematical formula that the system can effectively maintain.

In a shared memory system, the Magic Memory programming model treats these specifications on regions of memory as an invariant that is constantly maintained. Input Magic Memory regions store data used as input to the Magic Memory function, while output Magic Memory regions store the function's output based on the input region's values. When new data is written to the input memory region(s), the values in the output memory region(s) are updated to maintain the invariant. By describing the computation between memory regions, the input and output data can be stored in various places across the system, and this complexity can be hidden from the user. The mathematical basis of the function describing the relation between the memory regions allows a code generation system, like SPIRAL [62], to understand the computation and generate a program or hardware for a wide range of architectures.

The programmer's specification over a region declares that the value of a function is held over one or more other regions. Following this each memory region is accessed with ordinary loads and stores. There is no schedule, no kernel launch and no synchronization that needs to occur in the program. These operations are handled behind the memory interface. A store to an input region is what obliges the output to change and a load from the output region is when the computation is required to correct.

Maintaining the relationship between memory regions is now owned by the system instead of the programmer. This is manifested with three key responsibilities of the system. The first is that the system owns the physical representation of the each memory region and this does not need to mirror the logical view that is seen from the program. This is especially apparent when working with sparse data structures which can be logically viewed as a dense matrix while physically stored using any number of sparse data formats. The second responsibility is that the system is able to determine how to perform the computation that is declared by the invariant. As the specification between memory regions does not dictate the implementation, the system as the flexibility to perform this computation eagerly after a store to an input occurs or lazily by waiting until a load seeks to read the output value. Here the only constraint, is that the appropriate value is returned to the programmer. The last part of this is that the system is responsible for the placement and movement of the data. As the region can be located on whichever device in the system is best suited for the computation, the system is responsible for the data movement and transformations necessary in order to leverage the appropriate hardware.

```
1    int (*graph_mx)[NUM_NODES] = mx_malloc(sizeof(int[NUM_NODES][
     NUM_NODES]));
2    mx_set_property(graph_mx, MX_DEVICE_TYPE, MX_FPGA);
3    mx_set_property(graph_mx, MX_DIMENSION, MX_MATRIX);
4    mx_set_property(graph_mx, MX_DATA_TYPE, MX_INT);
5    mx_set_property(graph_mx, MX_SPARSE_TYPE, MX_CSR);
6    mx_set_property(graph_mx, MX_ROWS, NUM_NODES);
7    mx_set_property(graph_mx, MX_COLS, NUM_NODES);
8
9    float* pagerank_mx = mx_malloc(NUM_NODES * sizeof(float));
```

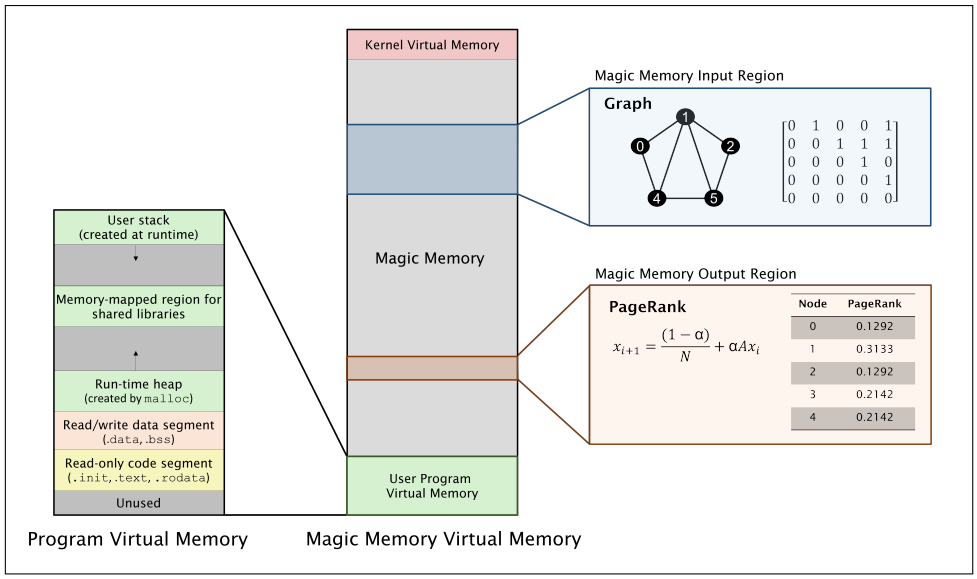

Figure 5.1: For x86 systems, Magic Memory regions utilize the virtual memory addresses outside of the user space and the kernel space. Here, the input Magic Memory region is a dense matrix which represents a graph while the output Magic Memory region is a vector with the PageRank of each node of the graph.

```
    mx_set_property(pagerank_mx, MX_DEVICE_TYPE, MX_FPGA);
    mx_set_property(pagerank_mx, MX_DATA_TYPE, MX_FLOAT);

    // Add some edges to the graph
    graph_mx[0][1] = 1;
    graph_mx[0][4] = 1;
    graph_mx[1][2] = 1;
    graph_mx[1][3] = 1;
    graph_mx[1][4] = 1;
    graph_mx[2][3] = 1;
    graph_mx[3][4] = 1;

    mx_setup(pagerank_mx, graph_mx, PAGERANK);

```

```
24     for(int i = 0; i < NUM_NODES; i++) {
25         printf("PageRank of Node %d: %f\n", i, pagerank_mx[i]);
26     }
27 
28     graph_mx[2][4] = 1;
29     printf("\nAdding edge from Node 2 to 4 to the graph\n\n");
30 
31     printf("Graph Adjacency Matrix\n");
32     for(int i = 0; i < NUM_NODES; i++) {
33         for(int j = 0; j < NUM_NODES; j++) {
34             printf("%d ", graph_mx[i][j]);
35         }
36         printf("\n");
37     }
38     for(int i = 0; i < NUM_NODES; i++) {
39         printf("PageRank of Node %d: %f\n", i, pagerank_mx[i]);
40     }
```

Listing 5.1: PageRank Algorithm using Magic Memory. Here the GraphBLAS Template Library (GBTL) [81] PageRank algorithm is used to describe the invariant between Magic Memory regions

Figure 5.1 illustrates how the virtual memory allocation of a Magic Memory program can exist in an x86 system, and Listing 5.1 shows a code snippet for this program. The user virtual memory space only uses the lower 48 bits of the 64 available bits in the x86 system with the kernel virtual memory only using a small portion of those upper bits. Magic Memory takes advantage of this and is allocated in the unused address space of x86 systems. This large address space that is available enables dense addressing for large sparse data structures. Then, as described later on, these addresses are able to be translated and utilize a wide range of sparse

data structures. Furthermore, since Magic Memory regions are allocated within the conventional virtual address space, these regions can be relocated across devices, allowing users to utilize various hardware accelerators with minimal change to the original program.

This program also demonstrates how key information regarding each region of memory is determined. Each memory region maintains key metadata in order to understand the data structure that is being stored there. This includes information on where the data should reside, the dimensionality of the data, data type, sparse format to use if any and the size of each dimension. Here, it should be noted that the programmer is not necessarily responsible for specifying every property. If properties are are not outlined by the programmer, default values are assumed based on the known information. For instance, if the size of a dimension is not known, the data type and number of bytes used to allocate the memory region can be used to infer the length of a vector. Other attributes of the API and compute stack are outlined in the next section.

5.2 Magic Memory API

Magic Memory is made accessible to the programmer through a simple header-only C library designed to mimic the C standard library as seen in Table 5.1. The input and output Magic Memory regions are allocated using Magic Memory equivalent versions of the `malloc` and `calloc` functions. This function returns an illegal address by taking advantage of the fact that x86_64 virtual address space only uses 48 of the 64 available bits. The extra 16 bits are then used to identify each Magic Memory region that is allocated, with each region having 256TB of addressable memory. This enables Magic Memory to support a dense address space for very large sparse

data structures. While the `calloc` function allows the data to be initialize with all 0s, the `memcpy` function allows for bulk data transfers so a Magic Memory region does not necessarily need to use individual loads and stores for initialization. Here, it can be noted that these operations do not necessarily allocate the memory that the program asks for. As the system is able to own the physical representation of the data, the required allocation for a region of memory is not required to happen until a load or store occurs. This enables the system to wait until the programmer describes other properties of the memory region such as the device and sparse data format before performing any memory allocation. Following this, any attempt to read or write a Magic Memory address triggers a segmentation fault and invokes a custom segmentation fault signal handler. This handler reads the 16 most significant bits in order to identify which Magic Memory region is being read. The handler goes on to decode the violating instruction and extracts relevant information, including the type of memory transaction (read or write), the address being accessed, and the register to store the result (for read instructions).

An invariant is declared as an expression in a domain-specific library for linear algebra. An expression written against BLAS [82] or GraphBLAS [83] carries its operand shapes explicitly, from which the dimensionality of the output region and its dependence on the inputs follow directly. The declaration therefore supplies everything the system requires, and nothing below the invariant table depends on how the expression was written. The dependence tracking and the representation of each region can be determined based on the knowledge of what the computation requires. The implementation described here does not yet include this front end. The invariant type is drawn from a fixed enumeration, and the system dispatches on it to select a maintenance routine for a given pair of regions, with each routine

Table 5.1: The Magic Memory API. Allocation and initialization mirror the C standard library; the remaining functions declare invariants between regions and configure their representation.

Function	**Description**
Allocation and initialization	
`mx_malloc`	Allocates a Magic Memory region of the given size and returns a non-canonical address encoding its region identifier.
`mx_calloc`	As `mx_malloc`, for an array of elements, with the region zero-initialized.
`mx_memcpy`	Copies data from a conventional address into a Magic Memory region, initializing the region's backing representation and metadata.
`mx_free`	Releases a region and its associated metadata.
Declaring computation	
`mx_setup`	Declares an invariant of the given type between a source region and a destination region, recording the dependence in the invariant table.
`mx_set_property`	Sets a property of a region — the device on which it resides, its element type, its dimensionality, its sparse representation, or its extents.
Representation access	
`mx_get_CSR_view`	Returns a view of a region's underlying compressed representation, for use by a backend that operates on it directly rather than through the logical interface.
`free_csr`	Releases a view obtained from `mx_get_CSR_view`.

written by hand. This is a demonstration of the layers beneath the table rather than of the table's population, and the enumeration should be read as standing in for the expression that a domain-specific front end would produce.

Key metadata on each Magic Memory region is stored in a struct that is created when each region is allocated. This metadata as shown in Table 5.2 contains information regarding the size, whether the region is sparse, where this data is actually stored, which other Magic Memory regions depend on the values from this region, and what the relationship is between the Magic Memory regions. Upon

Table 5.2: Region properties set through `mx_set_property`.

Property	Values	Meaning
MX_DEVICE_TYPE	MX_CPU, MX_FPGA, MX_PIM, MX_RRAM	Where the region's data resides and where its invariant is maintained.
MX_DATA_TYPE	MX_INT, MX_FLOAT	Element type.
MX_DIMENSION	MX_SCALAR, MX_VECTOR, MX_MATRIX	Logical shape of the region.
MX_SPARSE_TYPE	MX_DENSE, MX_CSR, MX_COO	Physical representation used for the region's backing store.
MX_ROWS	integer	Number of rows in the logical view.
MX_COLS	integer	Number of columns in the logical view.

each Magic Memory load or store that occurs, a segmentation fault occurs due to the illegal address that is being accessed. When the Magic Memory signal handler decodes the index and region that the original program attempted to write, the system references the corresponding struct to glean enough information to update the appropriate sparse or dense data structure and can propagate this information across the system as needed.

5.3 Magic Memory Consistency

By introducing implicit invariants between distinct regions of memory, the Magic Memory programming model breaks the assumptions of traditional memory consistency models. Because writes to an input magic memory region implicitly alter the state of a corresponding output region, the precise temporal ordering of load and store transactions becomes critical to program correctness.

In traditional architectures, performance is heavily based on reordering independent memory transactions. This reordering occurs at two distinct levels, both of which pose challenges for magic memory regions. First, at the hardware level,

out-of-order (OoO) execution pipelines and memory disambiguation predictors dynamically reorder loads and stores to different physical addresses to hide latency. Because the CPU hardware is unaware of the spatial dependency between a Magic Memory input and output region, it may incorrectly issue a load to the output region before the requisite store to the input region has committed to memory. Second, at the software level, optimizing compilers statically schedule instructions and frequently reorder memory accesses at compile time. Because the input and output regions reside at different addresses, the compiler cannot recognize the dependency between them. Consequently, optimizations can invert the program order of these dependent memory transactions.

To prevent both hardware and software from reordering these transactions, explicit synchronization is required. Memory barriers (fences) and compiler directives must be inserted between accesses to dependent input and output regions. While a sequence of independent loads or stores to a single magic memory region can still be freely reordered, any transition between interacting regions requires a strict barrier. This ensures that all preceding stores to an input region are globally visible and that compiler instruction scheduling remains consistent with the program order before subsequent dependent loads are issued.

5.4 Sparse Data Structures

Magic Memory presents a dense access interface to data that is physically stored in a compressed form, and assumes responsibility for maintaining that compressed representation as the program reads and writes it. By hiding the sparse data format from the programmer, the system introduces a clean separation between the application description and the underlying hardware implementation.

5.4.1 Dense Interface for Sparse Data

A program addresses a Magic Memory matrix as though it were a dense two-dimensional array, using the familiar double-subscript form `A[i][j]`. Assignment to a position that is not currently stored is a legal operation and causes the entry to appear as normal and a read of such a position returns zero. The compressed representation is never named in the source program, and no code manipulates row pointers or column indices of the sparse format directly. This behavior is illustrated in Figure 5.2 where the logical view from the programmer's perspective differs from the representation in hardware.

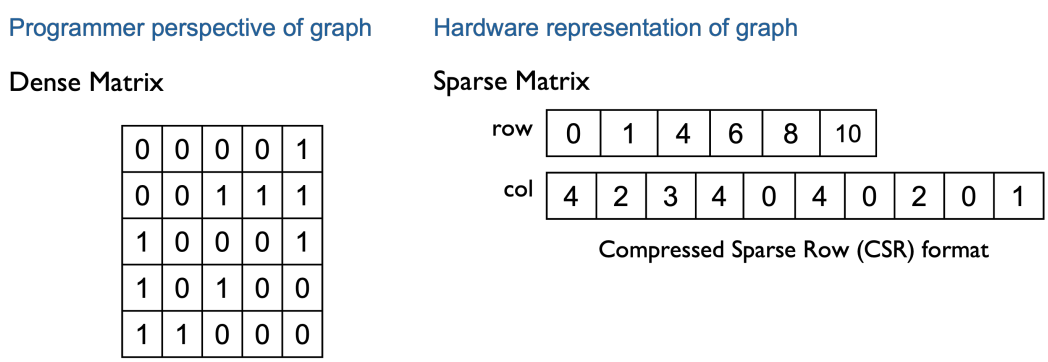

Figure 5.2: Dense versus sparse view of a matrix as seen from Magic Memory. Logically from the programmer's perspective, the entire matrix is available and stored in memory but Magic Memory hides the compressed physical representation that only tracks the nonzero elements.

A graph of $|V|$ vertices can be logically viewed as an adjacency matrix $\mathbf{A}$ of dimension $|V| \times |V|$ in which a_{ij} is nonzero when an edge runs from i to j. Inserting the edge (i, j) is simply the assignment `A[i][j] = w`, and deleting it is the assignment `A[i][j] = 0`. No separate mutation interface is required, and no distinction is drawn between a graph that is being modified and one that is being read. Traversal of a vertex's neighborhood, correspondingly, is a simple iteration

over `A[i]`. This manner of describing graph algorithms will be relevant as section 5.7 develops PageRank as a case study in which both paths are exercised.

5.4.2 Physical Representation and Format Selection

Abstracting the data structure is critical because there exists a wide range of sparse formats and a substantial body of work on tailoring them to both the structure of the sparsity and the computation the data will feed. Coordinate (COO) storage keeps an explicit triple for every nonzero and is trivial to build but wasteful to traverse. Blocked and vectorized formats such as BCSR and ELLPACK trade a controlled amount of explicit zero padding for regular access patterns that suit wide SIMD units. Higher-dimensional generalizations, such as compressed sparse fiber, extend the same principle to tensors. A parallel line of work treats format selection itself as a first-class problem, either by choosing among formats at runtime based on measured properties of the input, or by generating format-specific code from a declarative specification of the computation [84–86].

5.4.3 Managing the CSR Backend

For Magic Memory, the Compressed Sparse Row (CSR) format is utilized as the standard backend representation for sparse matrices. An $m \times n$ matrix $\mathbf{A}$ with nnz nonzeros is stored as three arrays. There is a value array and a column-index array, each of length nnz, holding the nonzeros of $\mathbf{A}$ in row-major order, and a row-pointer array of length $m + 1$. Entry i of the row-pointer array gives the index at which row i begins in the other two arrays, so that row i occupies the half-open range $[\mathrm{rowptr}[i], \mathrm{rowptr}[i+1])$ and $\mathrm{rowptr}[m] = nnz$. Row lengths are therefore implicit in the differences between adjacent pointers, and the nonzeros of a row are contiguous

Interface operation	Effect on the CSR representation	Cost
Read `A[i][j]`, stored	Search row i for column j	$O(\log nnz_i)$
Read `A[i][j]`, not stored	Search row i, return 0	$O(\log nnz_i)$
Write nonzero, stored	Update the value array in place	$O(\log nnz_i)$
Write nonzero, not stored	Insert; shift arrays, adjust row pointers	$O(nnz)$
Write zero, stored	Delete; shift arrays, adjust row pointers	$O(nnz)$
Iterate `A[i]`	Sequential scan of row i	$O(nnz_i)$

Table 5.3: Operations on the Magic Memory dense interface and their effect on the underlying CSR representation. nnz_i denotes the number of stored nonzeros in row i.

in memory.

That contiguity is what makes CSR fast to traverse and what makes it, in practice, immutable. Reading row i is a sequential scan of two arrays, which perfectly matches the streaming behavior that PIM systems depend on. Writing a value that is already stored is equally cheap, since it touches one element of the value array and leaves the structure untouched.

However, writing a nonzero into a position that is currently a structural zero requires shifting every subsequent entry of the value and column-index arrays by one and incrementing every row pointer above i. A single-element write costs $O(nnz)$. This asymmetry is severe enough that standard practice traditionally avoids the operation entirely: sparse structures are usually built once in a mutable format such as COO, converted to CSR, and thereafter treated as read-only.

Table 5.3 enumerates what each operation on the dense interface costs against the underlying CSR representation. The essential observation is that while the programmer's interface is uniform, the underlying physical costs are highly non-uniform. A read of a structural zero is indistinguishable to the program from a read of a stored nonzero, and returns 0 in both cases, but the search that produces the

answer is performed either way. As shown in the table, two entries are asymptotically worse than the rest because they induce structural changes rather than simple value updates. Deletion is the mirror of insertion. Assigning zero to a stored position removes the entry outright as the value and column-index arrays are shifted down by one, and every row pointer above i is decremented. Magic Memory stores no explicit zeros.

5.5 Graph Degree

Consider computing the degree of every vertex in a graph. Expressed against a dense adjacency matrix, the degree is a reduction over a row,

$$d_i = \sum_{j=0}^{|V|-1} \left[\, a_{ij} \neq 0 \,\right],$$

and a program written in those terms must examine every entry of the row. The materialized view of Section 5.4.1 improves on this by visiting only the stored entries, reducing the cost of the full degree distribution from $\Theta(|V|^2)$ to $\Theta(nnz)$. Neither is necessary, because neither needs to happen at query time at all.

Rather than computing d from A when it is asked for, a program declares that d stands in a fixed relationship to A and leaves the memory system to keep it true:

```
int (*graph_mx)[NUM_NODES] = mx_malloc(sizeof(int[NUM_NODES][NUM_NODES
    ]));
mx_set_property(graph_mx, MX_DIMENSION,   MX_MATRIX);
mx_set_property(graph_mx, MX_SPARSE_TYPE, MX_CSR);
mx_set_property(graph_mx, MX_ROWS, NUM_NODES);
mx_set_property(graph_mx, MX_COLS, NUM_NODES);
```

```
6
7 int *graph_degree_mx = mx_malloc(NUM_NODES * sizeof(int));
8
9 mx_setup(graph_degree_mx, graph_mx, INV_21);
10
11 graph_mx[0][3] = 1;
12 graph_mx[1][3] = 1;
13 graph_mx[2][3] = 1;
```

After the three stores, `graph_degree_mx` holds the degree of every vertex. No traversal was written, no reduction was invoked, and no view was taken.

5.5.1 Differential Maintenance

The graph degree algorithm provides an example of an invariant admits a differential update. When a store changes a_{ij} from v_{old} to v_{new}, the handler has in hand the previous value of the entry, the value being written, and the current value of the affected outputs. The correction to the degree follows from those three quantities alone:

$$\delta = \big[\, v_{\text{new}} \neq 0 \,\big] - \big[\, v_{\text{old}} \neq 0 \,\big], \qquad d_i \mathrel{+}= \delta, \qquad d_j \mathrel{+}= \delta, \tag{5.1}$$

with $\delta \in \{-1, 0, +1\}$ and both endpoints updated because the graph is undirected. An insertion into an empty position contributes $+1$, a deletion contributes -1, and a store that overwrites a nonzero with another nonzero contributes nothing and touches no output at all.

Three properties of (5.1) are what make the maintenance cheap enough to perform eagerly upon a store to the graph. The work is constant per store and independent of the degree of either endpoint, of nnz, and of $|V|$. The correction reads no part of the adjacency structure beyond the entry being written, so no traversal is

triggered and no additional rows are brought into cache. The two outputs it touches are single words, so a store to the graph becomes a store to the graph plus at most two scalar read-modify-writes.

While graph degree provides one such example of an invariant that can be transformed into differential compute, many more operations are possible. Invariants that can be expressed as linear transformations are possible as due to linearity $f(a + \delta) = f(a) + f(\delta)$. This means that by tracking the change to magic memory input regions upon a store, the new output value can be computed just from evaluating the function based on the delta. In the current Magic Memory prototype, the differential handlers that execute Equation 5.1 are systematically defined by the runtime. However, because the invariant is declared algebraically, it is theoretically possible for a code-generation system to automatically derive these differential update kernels directly from the programmer's high-level specification. Developing the compiler infrastructure to automatically synthesize these differential hardware handlers remains an active area of future work.

5.6 Prototype Magic Memory System

The Magic Memory programming model is initially prototyped using a CPU-FPGA system. This setup serves as a proxy for a memory-centric system, establishing a shared memory environment between the host processor and a near-memory accelerator. The evaluation system consists of an Intel i7-10700 CPU running CentOS 6, connected to a Stratix 10 MX FPGA via PCIe Gen 3. In this prototype, the FPGA is responsible not only for computing the invariants between Magic Memory regions, but also for managing the physical storage of the input and output data.

The system that is used to prototype Magic Memory can be seen in Fig-

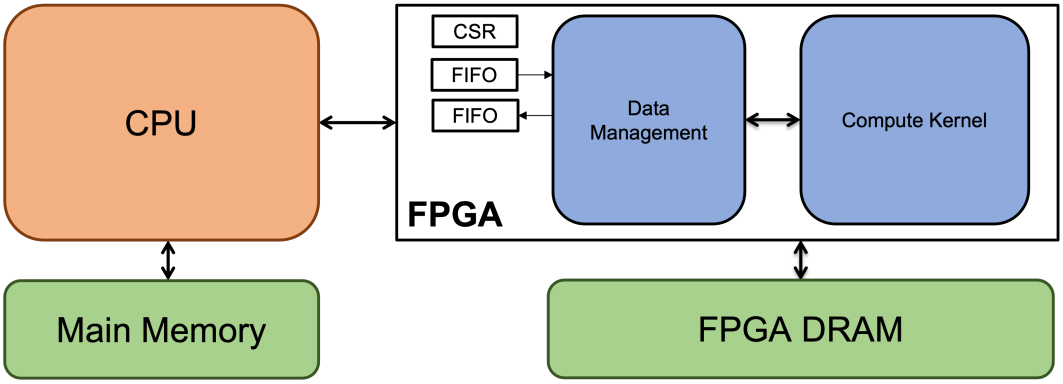

Figure 5.3: Magic Memory on CPU-FPGA System.

ure 5.3. The CPU and FPGA are connected via PCIe and each possesses its own main memory. On the FPGA, two key hardware acceleration modules exist, a data management module for sparse data structures and a compute kernel for maintaining an invariant. Here control status registers (CSRs) serve to track the appropriate metadata regarding problem parameters and the data within each memory region. An input and output FIFO are present for receiving and responding to load and store transactions that are redirected to the FPGA from the CPU.

To evaluate the feasibility of this programming model, we first measure the base cost of the abstraction layer. Because this prototype utilizes a standard PCIe connection rather than a native cache-coherent interconnect (such as CXL), the hardware cannot natively trap and route memory accesses. Instead, this system emulation relies on the OS-level signal handler as described in Section 5.4.1. When the host CPU issues a load or store to a Magic Memory region, it triggers a page fault. The software handler intercepts this fault, decodes the memory instruction, and manually routes the transaction to the FPGA by writing it to a memory-mapped FIFO over the PCIe bus. Consequently, every Magic Memory access incurs a software context-switch penalty. Table 5.4 quantifies this abstraction overhead by

comparing a direct memory-mapped FIFO access against an access routed through the Magic Memory signal handler framework.

	Read (μs)	Write (μs)
Memory-Mapped Access	5.46	0.54
Magic Memory	6.6	1.57
Difference	1.14	1.03

Table 5.4: Memory Mapped FIFO Latency

To quantify this abstraction overhead, a latency test was conducted to measure one million consecutive read and write accesses to the FPGA's memory-mapped FIFO. As shown in Table 5.4, routing these accesses through the Magic Memory programming model adds approximately 1.0 to 1.1 microseconds of latency per transaction compared to direct FIFO writes. On the evaluation system, this translates to an overhead of approximately 3,000 CPU cycles required to trap the memory address and execute the abstraction logic.

Although this software penalty is non-trivial, it is comparable in magnitude to other severe architectural memory stalls, such as a Translation Lookaside Buffer (TLB) miss or a lower-level cache miss. More importantly, this 3,000-cycle overhead is largely an artifact of this specific user-space emulation rather than a fundamental limitation of the programming model. The bulk of this latency stems from the expensive context switches required to move the CPU between user mode and kernel mode when invoking the signal handler before the program can resume.

This overhead can be dramatically reduced in future iterations by utilizing hardware interrupts instead of POSIX signals, which would bypass much of the OS-level routing and avoid the expensive switching between operating modes. Alternatively, booting and running the machine directly in kernel mode would eliminate

the user-to-kernel context-switching penalty entirely. While operating exclusively in kernel mode is not ideal from a security and isolation standpoint, it provides a viable architectural workaround to solve the context-switching bottleneck during prototype evaluation.

5.7 Magic Memory Example: PageRank

As a first example, this programming model is realized for the popular PageRank [87] algorithm using an FPGA. In the PageRank algorithm, the graph nodes are assigned a rank based on the number of edges coming in and out of each node. The graph is allocated using a Magic Memory input region, the ranking for each node is allocated to a Magic Memory output region, and the relation between the two regions is defined as the PageRank algorithm. This enables data analysts to observe the effects of adding or removing an edge from the graph from a single store to memory. In the magic memory input region, the sparse nature of the graph is hidden from the programmer, and a single load or store to memory is needed to update the graph data structure. The magic memory output region is allocated on the FPGA allowing for a specialized sparse matrix vector multiplication (SpMV) accelerator to be leveraged to perform the primary computation in the PageRank algorithm. In this example, the algorithm implemented using Magic Memory benefits because of its iterative nature, the stream of updates to a sparse data structure, and the frequency of these updates.

PageRank is a common algorithm used by search engines to rank web pages in search results. This algorithm uses the links coming to and from each page and the quality of the pages that the links come from in order to create the ranking. In PageRank, the input region is the graph of hyperlinks in matrix format with each

element being the probability of jumping from node to node. An important property of a hyperlink matrix is that the sum of every column must be one. Therefore if a node (column) has no outgoing edges (elements within a column), that column will have a uniform distribution such that this property is maintained.

The PageRank algorithm then iteratively performs a **Sp**arse **M**atrix dense **V**ector (SpMV) multiplication. The matrix representing the graph is multiplied by the current PageRank and the PageRank is updated with the output of this operation. This process is repeated until the difference between each iteration is less than some small epsilon. The output memory region stores the rank of each node in the input graph. While SpMV is a computation common in many graph applications, efficient SpMV is difficult due to the large amount of accesses to random memory locations. For this reason, a high performance hardware accelerator on the FPGA would greatly improve the performance of this application along with many more.

With Magic Memory, the graph is allocated as a matrix in a Magic Memory input region and the PageRank vector is allocated as a Magic Memory output on an FPGA. When writing the Magic Memory input region, the memory address is broken down to determine the row and column index of the data access. The row and column index are then used to index into the sparse data structure that is used to store the graph. By hiding the sparse data format from the programmer, this introduces a level of separation between the application description and the software or hardware implementation that is used behind the scenes in order to achieve good performance.

The core computation of PageRank (SpMV) is implemented in hardware on the FPGA to take advantage of the sparse data format. The Compressed Sparse Row (CSR) representation of the matrix, the read pattern of the data and column

arrays follows a simple sequential pattern which lends itself to using a burst coalesced Load-Store Unit (LSU). This LSU buffers contiguous memory requests for the largest possible burst. In order to utilize this hardware, each row is stored as a chunk of elements of a fixed size with the excess entries simply set to 0. By using a set size, the loop that iterates over the nonzeros of each row is able to be fully unrolled at compile time with no extra control logic inside the loop needed to check to see if the end of the row has been reached. This also allows for the LSU selected to have a bandwidth that matches the number of entries needed to fill the fully unrolled loop. Since each element of the output vector can be computed independently, the loop that iterates over these elements is pipelined and a loop iteration is able to be scheduled every cycle.

5.8 Evaluation

The FPGA being targeted for this application is a Stratix 10 PAC card with 32 GB of DDR4 memory. As a comparison to Magic Memory, we use a CPU implementation that also computes the SpMV using the CSR representation of the matrix. This program is run on an Intel Xeon Platinum 8256 as a baseline comparison.

Various different types of datasets were used to evaluate the FPGA design. The first example utilizes a graph with 100k nodes, where each node has 32 edges. This graph is constructed in such a way that it perfectly fits the hardware generated for the FPGA and serves to provide an upper bound on the performance that can be expected from hardware. Following this, other graphs with other average degrees are used to investigate how this kernel fairs for graphs that do not always use the maximum bandwidth to load useful data. In Figure 5.4, the runtime of SpMV for various different graph sizes is shown. Due to the nature of the hardware, the

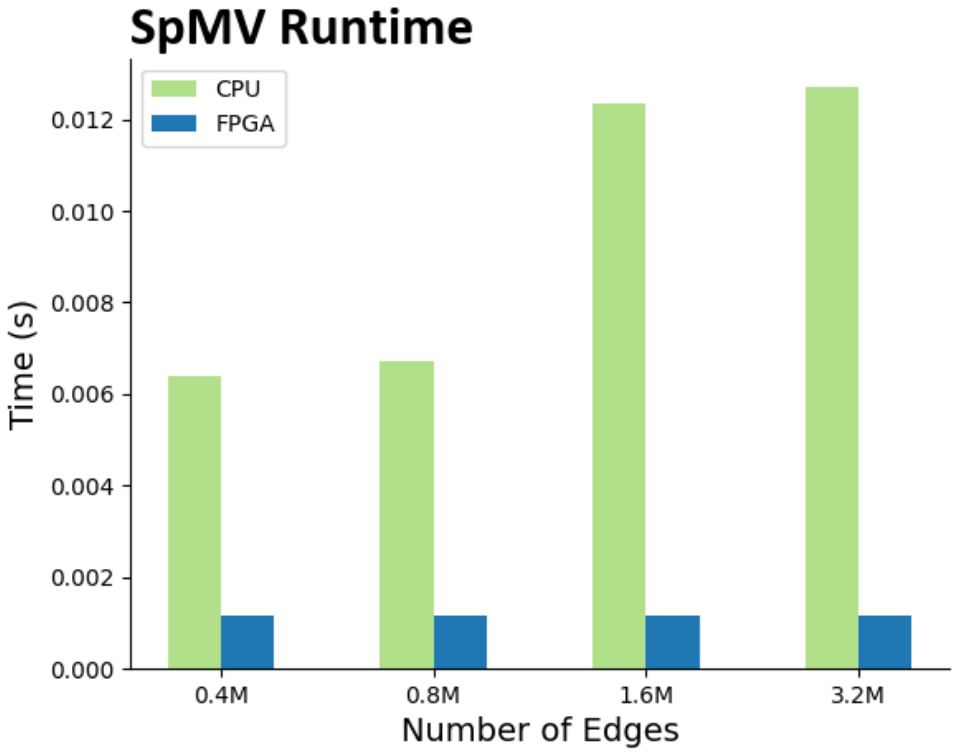

Figure 5.4: Comparison of SpMV kernel on FPGA versus baseline CPU implementation. All graphs used to generate the sparse matrix have 100,000 nodes which makes the SpMV operation a 100k x 100k sparse matrix multiplied by a 100k dense vector. By varying the number edges in graph, the sparsity of the matrix is changed

number of elements in each row of the matrix does not affect the running time of the FPGA. For this reason, for a graph where the maximum number of edges for each node is less than the block size (32 in this case) and a set number of nodes, the runtime is constant. This can be observed in the figure where the blue bars are a constant height. For the CPU implementation, the average degree of the graph is directly correlated with the runtime of the graph. However, even with graphs of an average degree of 4 the FPGA implementation is still able to provide a near 6x improvement over the baseline CPU implementation.

These performance results on the prototype CPU-FPGA system serve as a proof of concept that a declarative specification over regions of memory carries sufficient information for the system to automate data orchestration and leverage specialized hardware accelerators. The 6x performance improvement observed in Figure 5.4 represents a direct comparison of the isolated SpMV kernel executing on the FPGA versus the CPU baseline. While this measurement strictly evaluates the kernel performance rather than the end-to-end application overhead, it illustrates how the Magic Memory abstraction can be utilized. From the programmer's perspective, the graph is interacted with through a dense, two-dimensional interface. The programmer writes no instructions for resizing the CSR format used to insert or remove edges and does not need to define burst-coalesced Load-Store Unit (LSU) configurations. As the declarative specification defines the invariant between the regions, the runtime is granted the flexibility to physically represent the data in CSR and route the heavy computation to a highly optimized, format-specific SpMV kernel on the FPGA.

Furthermore, this proof of concept demonstrates that the system can assume responsibility for dynamic orchestration. In the interactive PageRank scenario, where edges are continuously added or removed, the programmer simply performs standard memory stores to update the graph. The Magic Memory system is responsible for recognizing that the graph has been modified and invariant requires the recomputation of the PageRank vector. When this recomputation is triggered, the system seamlessly invokes the accelerated SpMV kernel without any explicit synchronization or kernel-launch directives from the host program. This demonstrates that for memory-centric systems equipped with a shared address space, delegating the physical representation and orchestration to the system allows programmers to

seamlessly benefit from specialized hardware while focusing solely on the high-level mathematics of their application.

5.9 Summary

In this chapter, the Magic Memory programming model was presented for memory-centric systems with a shared address space and general purpose compute capability. In this model, computation is expressed by declaring persistent mathematical invariants between regions of memory, which the system automatically maintains. As the system understands the logical mathematical object within a region of memory, the system is able to manage the physical representation. This abstraction proves especially useful for managing dynamic sparse data structures where elements must be inserted and removed.

The declarative specification of the invariant allows the system to manage how and when computation occurs. The system can relocate computation to near memory accelerators and manage whether the computation occurs lazily or eagerly upon loads or stores to a memory region. When applicable, the system can further optimize execution by performing a differential computation to incrementally update the outputs as the input memory regions change. To validate this programming model, this was prototyped on a CPU-FPGA system with an interactive PageRank application. This example demonstrates how describing computation using a specification between regions of memory enables the system the ability to efficiently orchestrate data movement, manage sparse representations, and determine how and when computation occurs.

Chapter 6

Conclusion

For memory-centric systems, the diversity and difficulty in programming these systems have made them difficult to adopt for domain experts interested in applications that could serve to benefit from them. Most existing APIs and approaches require developers to write a program specific to a single hardware target which rigidly describes the number of processing units, scratchpad capacity, transfer granularity and communication topology. Meanwhile, none of these features are necessarily transferrable between hardware targets or next generation systems. For domain experts interested in sparse linear solvers or graph algorithms where the data movement requirement serves as the performance bottleneck, using these memory-centric systems without abstractions becomes a difficult and constantly evolving task. This dissertation asserts that the appropriate level of abstraction for these memory-centric systems comes from expressing computation as a declarative specification on regions of memory where key operations identified from the algorithm also map well to the machine.

To support this thesis, this dissertation presents two key programming mod-

els for memory-centric systems. The first is the Filter-Update model for capability-limited PIM architectures. In iterative algorithms commonly used in optimization problems or for linear solvers, the elements of memory to be used to participate in each iteration can be selected using a filter. The update then applies an accumulation operation which enables the solution to progress closer to a solution. This pair of operations is not only common enough across a broad class of iterative numerical methods but also simple enough to be run on hardware with extremely constrained capabilities. Neither of these operations require a floating-point unit, hardware multiplier, or communication between processing elements and does not require many instructions. Neither operation is required to retain state beyond the element that is being filtered or updated. This simplicity allows this programming model to be agnostic to the hardware that is present in the system while dramatically reducing memory bandwidth bottlenecks.

The second is the Magic Memory programming model that is designed for shared-memory systems with more capable near-memory accelerators. In this model computation is described as a persistent mathematical invariant between regions of memory. As the system understands the mathematical object within a region of memory, it is able to manage dynamic sparse data formats and orchestrate data movement across devices. The invariant further carries enough information such that the system is able to have the flexibility to determine when and how computation occurs without explicit instructions from the programmer.

Specifically, this dissertation provides the following contributions:

- Identified filter and update as the primitives for capability-limited PIM. These two operations are expressive enough to carry a broad class of iterative numerical methods, yet cheap enough to run on hardware with no floating-point

unit and no communication between processing units.

- Demonstrated how iterative solvers can be co-designed with PIM hardware. Algorithms can be adapted to fit the Filter Update programming model for approaches that traditionally do require the entire dataset. This was highlighted by adapting Gauss-Seidel, a memory-bandwidth bound problem, using filter-update operations so that the filter update modified algorithm is still able to converge within a similar number of iterations.

- A characterization of the data-movement versus iteration-count trade-off. Filtering reduces per-iteration traffic but increases iterations to convergence, and the model pays only where the first effect dominates. Filtered gradient descent reduced cumulative data movement by 3.9× at 1.5× more iterations and filter-update modified Gauss-Seidel converged 3.81–6.29× faster across systems from 107,400 to 10.7 million unknowns. The characterization also locates where the trade-off inverts, as the filter selects higher-degree rows in late iterations.

- Declarative specification over memory regions is sufficient for managing data orchestration. From the Magic Memory programming model and the interactive PageRank application, this work shows that a mathematical description of computation between memory regions carries enough information for a system to determine what data must move and when. Individual loads and stores to memory are able to trigger computation eagerly or lazily and even differential compute can be used to maintain the correct state.

- Abstracted physical memory layouts behind a dense logical interface. By specifying what the data within a region of memory represents, the physical layout within memory can be hidden from the programmer. This is demonstrated

by using standard dense addressing to sparse matrix. Behind the logical interface, the data are represented using a compressed sparse row (CSR) format where the runtime silently manages the complexity of inserting and removing elements.

6.1 Lessons Learned

While developing the Filter-Update and Magic Memory programming models for scientific applications and graph analytics, several practical insights regarding the co-design of algorithms, software abstractions, and memory-centric hardware emerged:

Algorithmic co-design with the hardware is necessary. Memory-centric hardware cannot be treated as a simple accelerator for core kernels in any algorithm. Achieving performance requires mathematical co-design with the operations that the hardware is able to perform efficiently. For example, filtering gradient descent reduces per-iteration data movement but inherently increases the total number of iterations required to converge. Similarly, executing parallel PIM updates for Gauss-Seidel required introducing an upfront graph-coloring algorithm to safely manage dependencies and prevent divergence. Developers must be willing to trade off strictly following the original algorithm for near memory acceleration.

PIM must natively support both processing and memory. When attempting to utilize Processing-In-Memory devices to gain significant performance improvements, the hardware must seamlessly integrate both aspects without hidden data-movement penalties. For the vectors that are stored in filter or update regions, there are still instances in iterative algorithms where the entire or a different portion of the vector needs to be read for certain computation. For this reason, while PIM

hardware may enable loading or storing less data, it does not remove all need to read data from that region of memory. For many current devices, data must be moved between host memory and PIM-enabled memory, or undergo expensive layout translations (e.g., from interleaved to contiguous), before computation can even begin. For memory-bandwidth-bound applications, the entire goal of PIM is to eliminate data movement by computing where the data already resides. Furthermore, PIM approaches must not compromise standard DRAM transfer rates because operations like Filter-Update still require bulk data movement back to the host. Degraded outbound transfer bandwidth makes it difficult to surpass traditional baselines.

Native floating-point support is not a strict prerequisite. A common assumption is that the lack of native floating-point support in near-memory hardware is a dealbreaker for scientific applications. As shown in this work, this is not the case. By mapping only highly simplified operations (such as magnitude thresholding and basic updates) to PIM, alternative techniques like soft-float representations can be used effectively. The lesson is that PIM does not need to replace the host processor. PIM can focus on efficiently performing basic operations while leaving more arithmetically intensive operations for the more capable host hardware.

Declarative specifications on virtual memory are the appropriate abstraction. Finally, the most significant software lesson is that forcing programmers to manage architectural details—such as bank locations, transfer granularities, or scratchpad capacities—destroys portability. By shifting to declarative specifications over regions of memory, the programmer simply defines the mathematical invariants and operations. This delegates the complex, hardware-specific orchestration of data movement and physical layout to the underlying system, proving that hardware

capability, rather than programmer burden, should dictate how data are managed.

6.2 Future Work

This thesis has served as a demonstration on how memory-centric architectures can be made useful for algorithms that target large-scale, memory-bound problems. By demonstrating that operations as simple as filter and update can yield significant reductions in data movement, this research has showed that mapping complex scientific applications to capability-limited hardware can lead to performance improvements. Building on this, there remain several avenues for algorithmic, hardware, and software extensions.

6.2.1 Designing New PIM Hardware

From a hardware perspective, new microarchitectures can be designed in order to support filter and update operations even more efficiently in PIM hardware.

At the bank level, new processing units can be designed to more effectively match the internal DRAM bandwidth for these operations. For example, preliminary work has been done which places a SIMD core in which the width matches the global data line (GDL) that connects each bank. As seen in Figure 6.1, for an architecture where the GDL width is 128 bits and these operations are performed upon 32-bit data, a SIMD core with 4 lanes enables the processing units to match the bandwidth of the data that is read from each bank. The precise SIMD width here is not necessarily fixed and can be matched to the memory technology. For instance, while a 128-bit GDL uses a 4-lane 32-bit SIMD core, a GDDR implementation could utilize a 256-bit GDL matched with an 8-wide SIMD core.

To execute the filter and update algorithms, each SIMD core has three SIMD

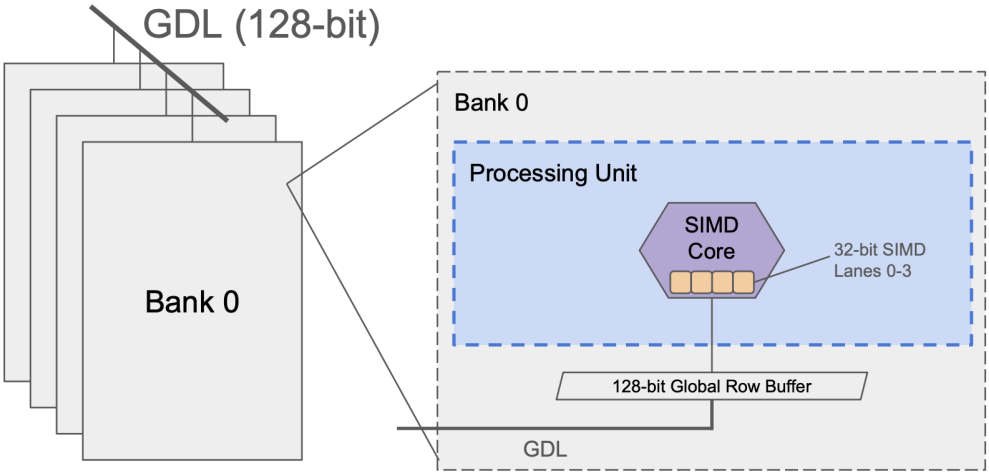

Figure 6.1: Potential bank-level PIM hardware for supporting Filter-Update operations. In this microarchitecture the SIMD width is matched to the size of the global row buffer and global data line (GDL) in order to support peak bandwidth.

registers for storing the operands and the output. As the dense data vector is distributed across multiple banks, each bank tracks the subset of the vector that it holds by storing the starting index of the first element in the bank within a dedicated register or within local memory. While the filter operation maps cleanly to the SIMD hardware, performing the update operation requires more consideration. As the update data are a sparse list of index value pairs, this requires writing indices that are scattered and non-adjacent. For this reason this design proposes a specialized set of instructions that restricts write accesses to the row buffer to only the first lane.

Future work looks to investigate microarchitectural enhancements to the bank-level processing unit itself. In particular, incorporating a dedicated local scratchpad memory into each bank's compute tile is being explored to buffer sparse outputs. By providing a local scratchpad to compact and organize filtered elements

before they cross the memory bus this seeks to further bridge the gap between internal DRAM bandwidth and sustained compute throughput. Other possibilities include exploring subarray-level PIM architectures. If subarray-level PIM can be engineered to efficiently support random accesses, it could yield even greater parallelism than the bank-level approach for these Filter-Update workloads.

Expanding Algorithmic Targets While this thesis has focused heavily on adapting sparse linear solvers for filter and update operations, it remains to be seen what other applications could fit this computational model. Future work must investigate how the Filter-Update model can be applied to more complex, hierarchical algorithms, such as algebraic multigrid. By working with domain experts who can leverage these operations to augment state-of-the-art approaches within their own specialties, it is highly likely that more algorithms can be adapted to reduce data movement. For instance, the Filter-Update methodology could be deployed not only to create sparsified smoothers within an algebraic multigrid hierarchy, but also to optimize the transitions between finer and coarser grids. Applying dynamic, magnitude-based filtering during the restriction (fine-to-coarse) and prolongation (coarse-to-fine) phases could significantly reduce the data movement required to project residuals and interpolate corrections across grid levels, unlocking further performance gains for massive scientific workloads.

High Bandwidth PIM Microarchitectures On the hardware front, current commercial PIM systems are heavily limited by the effective bandwidth they achieve, often struggling to reach the full throughput of the original memory interface. Future work remains to design new PIM microarchitectures capable of sustaining peak bandwidth for sparse workloads. From this, it remains to be seen what the true

performance ceiling for filter-update modified algorithms could be.

Semantic Capture of Magic Memory Invariants With respect to the Magic Memory programming model, approaches like LibraryX [63] enable using a high-level, domain-specific library to describe invariants between memory regions. LibraryX is a framework that can perform semantic capture by interpreting library calls as a specification. It creates an equivalent dataflow representation that is then passed into a code-generation system like SPIRAL. By deeply integrating this framework within Magic Memory, library calls, like those utilizing a GraphBLAS mathematical formulation, can serve as the declarative specification for the invariant. Code generation systems like SPIRAL [62] can then utilize this specification to automatically generate instructions optimized for whichever heterogeneous architectures are present within the system.

Automatic Derivation of Differential Updates Currently, Magic Memory invariants are described as explicitly defined relationships between memory regions. However, when an invariant is maintained eagerly, the system only needs to compute the differential update for the specific value that changed rather than recomputing the entire state. In these cases, future work remains on taking the declarative specification between regions of memory and automatically deriving what the exact differential update computation is and when this would be possible.

Bibliography

[1] I. S. Duff, R. G. Grimes, and J. G. Lewis, "Sparse matrix test problems," *ACM Trans. Math. Softw.*, vol. 15, p. 1–14, Mar. 1989. (document), 4.3.1, 4.4.2, 4.4

[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995. 1

[3] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "Ai and memory wall," 2024.

[4] S. K. Kingra, V. Parmar, C. C. Chang, *et al.*, "Slim: Simultaneous logic-in-memory computing exploiting bilayer analog oxram devices," *Scientific Reports*, vol. 10, p. 2567, 2020. 1

[5] K. Asifuzzaman, N. R. Miniskar, A. R. Young, F. Liu, and J. S. Vetter, "A survey on processing-in-memory techniques: Advances and challenges," *Memories - Materials, Devices, Circuits and Systems*, vol. 4, p. 100022, 2023. 1

[6] Y. Shin, J. Park, S. Cho, and H. Sung, "Pimflow: Compiler and runtime support for cnn models on processing-in-memory dram," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*,

CGO '23, (New York, NY, USA), p. 249–262, Association for Computing Machinery, 2023. 1

[7] J. Chen, J. Gómez-Luna, I. E. Hajj, Y. Guo, and O. Mutlu, "Simplepim: A software framework for productive and efficient processing-in-memory," 2023. 1, 1.1

[8] K.-U. Sattler and M. A. Jibril, "Tutorial: Programming processing-in-memory for data management," in *Datenbanksysteme für Business, Technologie und Web - Workshopband (BTW 2025)* (C. Binnig, R. Schenkel, A. Henrich, D. Nicklas, M. E. Schüle, and K. Meyer-Wegener, eds.), pp. 367–374, Gesellschaft für Informatik, Bonn, 2025.

[9] H. Kang, P. B. Gibbons, G. E. Blelloch, L. Dhulipala, Y. Gu, and C. McGuffey, "The processing-in-memory model," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, (New York, NY, USA), p. 295–306, Association for Computing Machinery, 2021. 1

[10] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," in *Proceedings of Machine Learning and Systems (MLSys)*, vol. 3, 2021. arXiv:2007.00072. 1.1

[11] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022. arXiv:2205.14135. 1.1

[12] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads,"

in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019. 1.1

[13] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 56–65, 2013. 1.1

[14] G. F. Oliveira, J. Gómez-Luna, L. Orosa, S. Ghose, N. Vijaykumar, I. Fernandez, M. Sadrosadati, and O. Mutlu, "DAMOV: A new methodology and benchmark suite for evaluating data movement bottlenecks," *IEEE Access*, vol. 9, pp. 134457–134502, 2021. arXiv:2105.03725. 1.1

[15] K. Asifuzzaman, Y. He, T. Zhang, E. Tang, N. Rao Miniskar, K. Teranishi, and J. S. Vetter, "A survey on the expanding scope and interdisciplinary opportunities for processing-in-memory techniques," *IEEE Access*, vol. 14, pp. 18408–18430, 2026. 1.1

[16] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison, "IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI," *IBM Journal of Research and Development*, vol. 62, no. 4-5, pp. 8:1–8:8, 2018. 1.1

[17] D. Cock, A. Ramdas, D. Schwyn, M. Giardino, A. Turowski, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, R. Achermann, G. Alonso, and T. Roscoe, "Enzian: An open, general, CPU/FPGA platform for systems software research," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 434–451, 2022.

[18] J. Alsop, M. D. Sinclair, and S. V. Adve, "Spandex: A flexible interface for efficient heterogeneous coherence," in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 261–274, 2018.

[19] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng, and O. Mutlu, "CoNDA: Efficient cache coherence support for near-data accelerators," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, pp. 629–642, 2019.

[20] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, "LazyPIM: An efficient cache coherence mechanism for processing-in-memory," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 46–50, 2017. 1.1

[21] F. A. Siddique, D. Guo, Z. Fan, M. Gholamrezaei, M. Baradaran, A. Ahmed, H. Abbot, K. Durrer, K. Nandagopal, E. Ermovick, K. Kiyawat, B. Gul, A. Mughrabi, A. Venkat, and K. Skadron, "Architectural modeling and benchmarking for digital dram pim," in *2024 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 247–261, 2024. 1.1, 2.1.2

[22] B. Hyun, T. Kim, D. Lee, and M. Rhu, "Pathfinding future PIM architectures by demystifying a commercial PIM technology," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 263–279, 2024. arXiv:2308.00846. 1.1

[23] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim, *et al.*, "25.4 a 20nm 6GB function-in-memory DRAM, based on HBM2 with a 1.2TFLOPS programmable computing unit

using bank-level parallelism, for machine learning applications," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 350–352, 2021. 1.1

[24] F. Devaux, "The true processing in memory accelerator," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–24, IEEE, 2019. 1.1, 2.1.2

[25] A. Devic, S. B. Rai, A. Sivasubramaniam, A. Akel, S. Eilert, and J. Eno, "To pim or not for emerging general purpose processing in ddr memory systems," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 231–244, 2022. 1.1, 2.1.2

[26] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. N. Vijaykumar, "Newton: A DRAM-maker's accelerator-in-memory (AiM) architecture for machine learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 372–385, 2020. 1.1

[27] D. Das Sharma, "Compute express link: An open industry-standard interconnect enabling heterogeneous data-centric computing," in *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, pp. 5–12, 2022. 1.1

[28] D. Das Sharma, R. Blankenship, and D. S. Berger, "An introduction to the compute express link (CXL) interconnect," *ACM Computing Surveys*, vol. 56, no. 11, pp. 290:1–290:37, 2024. arXiv:2306.11227. 1.1

[29] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu, "Enabling the adoption of processing-in-memory: Challenges, mechanisms, future research directions," 2018. 1.1

[30] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. D. Ferreira, N. M. Ghiasi, M. Patel,

M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, "Simdram: a framework for bit-serial simd processing using dram," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, (New York, NY, USA), p. 329–345, Association for Computing Machinery, 2021. 1.1, 1.2, 2.1.1

[31] M. S. Q. Truong, E. Chen, D. Su, L. Shen, A. Glass, L. R. Carley, J. A. Bain, and S. Ghose, "Racer: Bit-pipelined processing using resistive memory," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, (New York, NY, USA), p. 100–116, Association for Computing Machinery, 2021. 1.1

[32] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: in-memory accelerator for bulk bitwise operations using commodity dram technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), p. 273–287, Association for Computing Machinery, 2017. 1.2, 2.1.1

[33] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware architecture and software stack for pim based on commercial dram technology : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 43–56, 2021. 1.2, 2.2.2

[34] H. Lee, G. Kim, D. Yun, I. Kim, Y. Kwon, and E. Lim, "Cost-effective llm accelerator using processing in memory technology," in *2024 IEEE Symposium*

on VLSI Technology and Circuits (VLSI Technology and Circuits), pp. 1–2, 2024. 1.2

[35] UPMEM, "UPMEM - processing in-memory." http://www.upmem.com/, 2025. Accessed: 2025-12-08. 1.2, 2.1.2

[36] 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)*, pp. 105–117, 2015. 1.2, 2.1.2

[37] H. Ji, S. Vanavasam, Y. Zhou, Q. Xia, J. Huang, Y. Yuan, R. Wang, P. Gupta, B. Chitlur, I. Jeong, and N. S. Kim, "Demystifying a cxl type-2 device: A heterogeneous cooperative computing perspective," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1504–1517, 2024. 1.2, 4.5

[38] B. Friesel, M. Lütke Dreimann, and O. Spinczyk, "A full-system perspective on upmem performance," in *Proceedings of the 1st Workshop on Disruptive Memory Systems*, DIMES '23, (New York, NY, USA), p. 1–7, Association for Computing Machinery, 2023. 1.2

[39] J. Nutini, M. Schmidt, I. H. Laradji, M. Friedlander, and H. Koepke, "Coordinate descent converges faster with the gauss-southwell rule than random selection," 2018. 1.3.1, 3.2.1, 4

[40] A. Sahu, A. Dutta, A. M. Abdelmoniem, T. Banerjee, M. Canini, and P. Kalnis, "Rethinking gradient sparsification as total error minimization," in *Advances in Neural Information Processing Systems* (M. Ranzato, A. Beygelz-

imer, Y. Dauphin, P. Liang, and J. W. Vaughan, eds.), vol. 34, pp. 8133–8146, Curran Associates, Inc., 2021. 1.3.1, 4.2.3

[41] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A modern primer on processing in memory," *CoRR*, vol. abs/2012.03112, 2020. 2.1, 4.1

[42] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: in-memory accelerator for bulk bitwise operations using commodity dram technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), p. 273–287, Association for Computing Machinery, 2017. 2.1.1

[43] 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)*, pp. 481–492, IEEE, 2017. 2.1.1

[44] 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)*, pp. 383–396, IEEE, 2018. 2.1.1

[45] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 397–410, 2019. 2.1.1

[46] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 27–39, 2016. 2.1.1

[47] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016. 2.1.1

[48] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A DRAM-based reconfigurable in-situ accelerator," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 288–301, Oct. 2017. 2.1.2

[49] M. Lenjani, P. Gonzalez, E. Sadredini, S. Li, Y. Xie, A. Akel, S. Eilert, M. R. Stan, and K. Skadron, "Fulcrum: A simplified control and access mechanism toward flexible and practical in-situ accelerators," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 556–569, IEEE, 2020. 2.1.2

[50] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon, K. Kim, J. Jung, I. Yun, S. J. Park, H. Park, J. Song, J. Cho, K. Sohn, N. S. Kim, and H.-H. S. Lee, "Near-memory processing in action: Accelerating personalized recommendation with axdimm," *IEEE Micro*, vol. 42, no. 1, pp. 116–127, 2022. 2.1.2, 4.5

[51] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems," in *2016 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016. 2.1.2

[52] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: Experimental analysis and char-

acterization of a real processing-in-memory system," *IEEE Access*, vol. 10, pp. 52565–52608, 2022. 2.1.2

[53] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 283–295, 2015. 2.2.1

[54] S. Patel and W.-m. W. Hwu, "Accelerator architectures," *IEEE Micro*, vol. 28, no. 4, pp. 4–12, 2008. 2.2.1

[55] H. Kang, Y. Zhao, G. E. Blelloch, L. Dhulipala, Y. Gu, C. McGuffey, and P. B. Gibbons, "Pim-tree: A skew-resistant index for processing-in-memory," *Proc. VLDB Endow.*, vol. 16, p. 946–958, Dec. 2022. 2.2.1

[56] A. Lopes, D. Castro, and P. Romano, "Accelerating transactional execution via processing-in-memory," in *Proceedings of the 21st European Conference on Computer Systems*, EUROSYS '26, (New York, NY, USA), p. 1847–1861, Association for Computing Machinery, 2026. 2.2.1

[57] Y. Kwon, K. Vladimir, N. Kim, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, B. An, J. Kim, J. Lee, I. Kim, J. Park, C. Park, Y. Song, B. Yang, H. Lee, S. Kim, D. Kwon, S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, M. Lee, M. Shin, M. Shin, J. Cha, C. Jung, K. Chang, C. Jeong, E. Lim, I. Park, J. Chun, and S. Hynix, "System architecture and software stack for gddr6-aim," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, pp. 1–25, 2022. 2.2.1, 2.2.2

[58] 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), pp. 457–468, 2017. 2.2.2

[59] R. Liu, Z. Li, X. Zhang, X. Chen, Y. Han, and M. Tang, "Cim-blas: Computing-in-memory accelerator for blas," in *Proceedings of the 62nd Annual ACM/IEEE Design Automation Conference*, DAC '25, IEEE Press, 2025. 2.2.2

[60] W. Kahan, *Gauss-Seidel Methods of Solving Large Systems of Linear Equations*. PhD thesis, University of Toronto, Toronto, 1958. 2.3.3

[61] J.-Y. Yuan, "The ostrowski–reich theorem for sor iterations: extensions to the rank deficient case," *Linear Algebra and its Applications*, vol. 315, no. 1, pp. 189–196, 2000. 2.3.3

[62] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, "Spiral: Extreme performance portability," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018. 3.2.1, 3.2.2, 5.1, 6.2.1

[63] S. Rao, A. Prakash, N. Zhang, H. Mankad, and F. Franchetti, " LibraryX: A Framework for Cross-Library-Call Optimization ," in *2026 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, (Los Alamitos, CA, USA), pp. 1161–1175, IEEE Computer Society, May 2026. 3.2.2, 6.2.1

[64] K. Chmielewski, J. Ławnicki, U. Lukyanau, T. Kobus, and M. Maciejewski, "Upmem unleashed: Software secrets for speed," 2025. 3.2.4

[65] Y. Lan, L. Tang, N. Zhang, and F. Franchetti, "Double-precision floating-point on aws trainium using integer units: A first look," in *2026 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2026. 3.2.4

[66] NVIDIA, F. Abecassis, A. Agrusa, D. Ahn, J. Alben, S. Alborghetti, M. Andersch, S. Arayandi, A. Bjorlin, A. Blakeman, E. Briones, I. Buck, B. Catanzaro, M. Chang, J. Choi, M. Chrzanowski, E. Chung, V. Cui, S. Dai, B. D. Rouhani, C. del Mundo, D. Donia, B. Eryilmaz, H. Estela, A. Goel, O. Goncharov, Y. Guvvala, R. Hesse, R. Hewett, H. Hum, U. Kapasi, B. Khailany, M. Khona, N. Knight, A. Kondratenko, R. Krashinsky, B. Lanir, S. Layton, M. Lightstone, D. Lo, P. Micikevicius, A. Mishra, T. Moon, D. Narayanan, C. Ni, A. Paithankar, S. Pasumarthi, A. Patel, M. Patwary, A. Poojary, G. Prasad, S. Priyadarshi, Y. Qin, X. Ren, O. Rybakov, C. Sakr, S. Satheesh, S. Sergienko, P. Shamis, K. Shankar, N. Sharma, M. Shoeybi, M. Siu, M. Smelyanskiy, D. Stosic, D. Stosic, B.-Y. Su, F. Sun, N. Tajbakhsh, S. Thomas, P. Tredak, E. Tsykunov, G. Vaithilingam, A. Vavre, R. Venkatesan, R. Waleffe, Q. Wan, H. Wang, M. Wang, L. Wei, H. Wu, E. Wu, K. Wyss, N. Xu, J. Xue, C. Yang, Y. Zhai, R. Zhang, J. Zhu, and Z. Zhu, "Pretraining large language models with nvfp4," 2026. 3.2.4

[67] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture," 2022. 3.3.1

[68] D. Lee, B. Hyun, T. Kim, and M. Rhu, "Pim-mmu: A memory management unit for accelerating data transfers in commercial pim systems," 2024. 3.3.1

[69] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu, "Processing-in-memory: A workload-driven perspective," *IBM Journal of Research and Development*, vol. 63, no. 6, pp. 3:1–3:19, 2019. 4.1

[70] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication

on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–11, 2009. 4.1

[71] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, (New York, NY, USA), p. 273–282, Association for Computing Machinery, 2013. 4.1

[72] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2nd ed., 2003. 4.1, 4.4.2

[73] J. Nocedal and S. J. Wright, *Numerical Optimization*. New York, NY: Springer, 2nd ed., 2006. 4.1

[74] A. N. Sahu, A. Dutta, A. M. Abdelmoniem, T. Banerjee, M. Canini, and P. Kalnis, "Rethinking gradient sparsification as total error minimization," in *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual* (M. Ranzato, A. Beygelzimer, Y. N. Dauphin, P. Liang, and J. W. Vaughan, eds.), pp. 8133–8146, 2021. 4.2

[75] D. Alistarh, T. Hoefler, M. Johansson, S. Khirirat, N. Konstantinov, and C. Renggli, "The convergence of sparsified gradient methods," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, (Red Hook, NY, USA), p. 5977–5987, Curran Associates Inc., 2018. 4.2

[76] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra, "Matrix market: a web resource for test matrix collections," in *Proceedings of*

the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement, (GBR), p. 125–137, Chapman & Hall, Ltd., 1997. 4.3.1, 4.4.2

[77] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," tech. rep., University of Virginia, Charlottesville, VA, 1991–2007. Continuous updates. 4.4.2

[78] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial, Second Edition*. Society for Industrial and Applied Mathematics, second ed., 2000. 4.4.2

[79] G. Strang, *Computational Science and Engineering*. Wellesley, MA: Wellesley-Cambridge Press, 2007. 4.4.2

[80] J. H. Kim, S.-h. Kang, S. Lee, H. Kim, W. Song, Y. Ro, S. Lee, D. Wang, H. Shin, B. Phuah, J. Choi, J. So, Y. Cho, J. Song, J. Choi, J. Cho, K. Sohn, Y. Sohn, K. Park, and N. S. Kim, "Aquabolt-xl: Samsung hbm2-pim with in-memory processing for ml accelerators and beyond," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, pp. 1–26, 2021. 4.5

[81] "Graphblas template library (gbtl), version 3.0." Available at https://github.com/cmu-sei/gbtl, June 2020. 41

[82] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Softw.*, vol. 5, p. 308–323, Sept. 1979. 5.2

[83] J. Kepner, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, J. Moreira, P. Aaltonen, D. Bader, A. Buluc, F. Franchetti,

J. Gilbert, D. Hutchison, M. Kumar, and A. Lumsdaine, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, p. 1–9, IEEE, Sept 2016. 5.2

[84] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, "Taco: A tool to generate tensor algebra kernels," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 943–948, 2017. 5.4.2

[85] J. Li, G. Tan, M. Chen, and N. Sun, "Smat: an input adaptive auto-tuner for sparse matrix-vector multiplication," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), p. 117–126, Association for Computing Machinery, 2013.

[86] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing sparse matrix-vector multiplication using index and value compression," in *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, (New York, NY, USA), p. 87–96, Association for Computing Machinery, 2008. 5.4.2

[87] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120. 5.7