Rethinking Control, Access, and Communication Mechanisms for Data-Intensive Applications

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Marzieh Lenjani

May 2022

Abstract

In current computing systems, the transfer of data between the off-chip memory and the processor takes two orders of magnitude more time and consumes three orders of magnitude higher energy than the actual computation. This dissertation presents two approaches for alleviating the data movement overhead: (i) enabling flexible-bitwidth value representation in the memory hierarchy of general-purpose processors and (ii) processing data inside the memory.

In the first approach, we propose a narrow-bitwidth and overflow-free value representation technique in the cache and main memory of general-purpose processors, which provides 1.23× speed up compared to the state-of-the-art cache compression techniques.

In the second approach, we explore deviating from conventional Von Neumann architectures to minimize data movement by adding processing units as close as possible to the memory cells, i.e., processing in memory (PIM). We show that realizing the true potential of PIM requires rethinking and simplifying the control, access, communication, and load balancing mechanisms of PIM units.

Accordingly, first, we propose a PIM-based accelerator, Fulcrum, that introduces a new architecture for in-memory processing units. The architecture offers a trade-off between (i) full control and access divergence support in SISD and (ii) no/costly control or divergence support in SIMD/SIMT approaches. Fulcrum outperforms a server-class GPU with three stacks of HBM2 and, on average, provides 70× speedup per memory stack and reduces the energy consumption by 96%.

Next, we observe that rethinking communication and load balancing mechanisms can unlock the benefits of PIM for a wider range of applications. Accordingly, in the second version of Fulcrum, dubbed Gearbox, we add hardware support for (i) offloading accumulations operations from one memory segment to another memory segment, and (ii) balancing the load among processing elements. Gearbox can outperform Gunrock, a GPU-based graph-processing framework, by 15.73×. In the third version of Fulcrum, dubbed Pulley, we add hardware support that enables every group of processing elements cooperatively generate an intermediate array. Pulley, on average, delivers 20× speedup compared to Bonsai, an FPGA-based sorting accelerator.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Marzieh Lenjani

This dissertation has been read and approved by the Examining Committee:

Kevin Skadron, Adviser

Worthy N. Martin, Committee Chair

Ashish Venkat

Mircea Stan

Yangfeng Ji

Accepted for the School of Engineering and Applied Science:

Jennifer L. West, Dean, School of Engineering and Applied Science

May 2022

DEDICATION

I gratefully dedicate this dissertation to my love, Jalal, and to the memory of my father who always encouraged me to peruse science.

Acknowledgements

I am grateful to numerous people who have contributed to shaping this dissertation. This research would not have been possible without the influence, advice, and support of many colleagues, friends, and family.

First, I would like to acknowledge and give my warmest thanks to my advisor, Prof. Kevin Skadron, whose advice, support, and encouragement have had a tremendous impact on my professional and personal life. Kevin encouraged me to pursue non-incremental ideas, supported me even when facing setbacks, and challenged and helped me develop my ideas. I am very fortunate to learn how to approach research problems from one of the most influential scholars in computer architecture. I also could not bear the pain of losing my father during my Ph.D. studies without Kevin's support.

I also wish to thank the members of my Ph.D. committee; I had the pleasure to be a co-author with Prof. Mircea Stan and Prof. Ashish Venkat, and I am very grateful for what I learned from their valuable feedback and discussion. I would like to thank Prof. Worthy Martin and Prof. Yangfeng Ji for their many insightful discussions and suggestions on my research.

I also was fortunate to have many friends and collaborators, including Elaheh Sadredini, Reza Rahimi, Alif Ahmed, Farzana Siddique, Lingxi Wu, Patricia Gonzalez-Guerrero, Rasool Sharifi, Sergui Mosanu, Tommy Tracy, Vaibhav Verma, and Wole Jaiyeoba; I want to thank you all for all the enjoyable technical and non-technical conversations that we had.

Last but not the least, I would like to thank my family for their support throughout my entire life. I am grateful to my best friend and husband, Jalal, for his unconditional love and encouragement. I also wish to express my gratitude to my mother and brothers for their love and sacrifice.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Big data applications have enormous impact on many industries by extracting valuables information from data represented in large tables, graphs, etc. Similarly, machine learning, and particularly deep learning approaches based on artificial neural networks, has recently revolutionized many fields including computer vision, speech recognition, natural language processing, helping address the grand challenges facing our society.

Big data and machine learning applications increasingly require massive computing capacity, parallelism, and data movement between memory and processors, outpacing existing semiconductor technology. Year-over-year improvement in general-purpose CPU performance has been stagnating due to physical limits in processor technology (from doubling performance every 1.5 years (1985 through 2003) and two years (2003 to 2010) to now when the performance is expected to be doubled only every 20 years). This exacerbates the longstanding memory wall problem. Moving data to and from memory is becoming a huge bottleneck. For example, in current systems, the transfer of data, from the off-chip memory to the processor, takes two orders of magnitude more time and consumes three orders of magnitude higher energy consumption than the actual computation (e.g., a single-precision addition) [4]. Large working sets of some modern applications (with low temporal locality) or irregular access patterns (with low spatial locality) make caches (our traditional solution to the memory wall) less effective.

Recent integration technologies, such as 2.5D integration of memory and processing elements (e.g., integration of GPU/FPGA with HBM2 memories) have not completely solve the problem, and data movement still remains the main bottleneck for memory-intensive applications, where each loaded data elements from memory will only be the subject of a single or few compute operation. The overhead stems from the fact that memory is designed in a multi-level structure, and data should move along all the buses that connect

these structures. For example, each 3D memory stack has many memory layers (and sometimes a logic layer), each layer has many banks, each bank has many subarrays, each subarray has many rows, and each row is composed of many words (e.g., 64 words). When data are accessed, they traverse across all the buses and across the distance between the memory stack and the processing elements. To alleviate the cost of data movement of data-intensive applications, we have two options: (i) shrinking the size of data to reduce data transfer and (ii) processing data inside the memory by adding processing units near memory cells.

To reduce the cost of data movement in general-purpose processors, prior works have explored cache and memory compression to reduce the data transfer and also increase the effective capacity of on-chip and off-chip memory elements, which in turn reduces data movement (due to reduced cache misses and redcued memory page faults). We can categorize prior approaches into two groups: (i) lossless compression and (ii) lossy compression. Lossless compression impose significant overheads for conversion, metadata handling, and locating the compressed values. As a result, they are often only amenable for L3 [5, 6]. More importantly, they have low compression ratio for floating point values and cache blocks with multiple data types. The inefficiencies stem from the fact that compressor and decompressor have minimum information about the program and obliviously search for value locality within each cache block that they receive [5, 7]. Some prior works propose to trade off the accuracy for the size of transferred data by (i) quantizing floating point values or (iii) omitting (truncating) a certain number of least significant bits (LSBs) in the mantissa of floating-point numbers [1, 8] (we refer to these techniques as OLSB). These approaches can be considered as light-weight lossy compression technique. Quantization is popular in accelerators, where the bitwidth can be customized [9, 10, 11]. However, employing quantization in general-purpose processors is challenging. A software implementation of quantization imposes a significant conversion overhead and programmer's effort, and more importantly, cannot unlock maximum benefit for variables that require less bitwidth than standard variables. It has to use only 8-bit, 16-bit, or 32-bits variables and hence it imposes significant (e.g., 100%, 77%, or 88% for 4-bit, 9-bit, or 17-bit variables, respectively) cache space and memory bandwidth overhead. More importantly, the floating-point format and the exponent part are necessary for supporting a wide range of values. In fact, the floating-point format is popular among developers because it supports a wide range of values and decreases the probability of overflow during arithmetic operations. The other category of lossy compression techniques often keeps the exponent intact and reduces the bitwidth of values by truncating LSBs. Two factors limit the benefit of such techniques: (i) the overhead of eight bits for the exponent, and (ii) the high error that grows with the value of the exponent. In other words, the larger the magnitude of the value, the higher the absolute error (the magnitude of error).

The second option for reducing the cost of data movement is to add processing units closer to the memory cells, i.e., to subarrays so that the data is processed in situ. Processing at the subarray level

provides higher parallelism, lower access latency, and lower energy consumption compared to other levels of processing in memory. However, due to the small size of each subarray and the inefficiency of logic in memory technology, adding traditional cores per subarray or adding row-wide arithmetic units per subarray is impractical. Therefore, many of the existing subarray-level methods [2, 12] add only simple row-wide logic units to the row buffer of each memory subarray. In our paper, Fulcrum [13], we have shown that row-wide logic units fail to fully capitalize on the benefits of in situ processing because they require multiple energy-draining row activations. Furthermore, in previous in-situ techniques, the row-wide logic units perform the same operation on all the words in a row, making operation with data dependency and predicate-based operations inefficient. The inflexibility limits the range of operations that can benefit from in-situ computing.

1.1 Contributions

This dissertation explores two main research questions: (i) how can we enable light-weight narrow-bitwidth value representation and avoid overflows? (ii) how can we increase the flexibility and efficiency of in-memory processing units and widen the range of applications that can benefit from in-situ processing?

Accordingly, **we hypothesize that light-weight compression and flexible near-data processing can significantly mitigate the cost of data movement**. To evaluate this hypothesis, this dissertation proposes: (i) An Overflow-free Quantized Memory Hierarchy in General-Purpose Processors, (ii) Fulcrum: a Simplified Control and Access Mechanism toward Flexible and Practical In-situ Accelerators, (iii) Gearbox: A Case for Supporting Accumulation Dispatching and Hybrid Partitioning in PIM-based Accelerators, and (iv) Pulley: An Algorithm/Hardware Co-optimization for Multi-device In-memory Sorting.

1.1.1 An Overflow-free Quantized Memory Hierarchy in General-Purpose Processors

Accelerator designers exploit quantization to reduce the bitwidth of values and reduce the cost of data movement. However, any value that does not fit in the reduced bitwidth results in an overflow (we refer to these values as outliers). Therefore accelerators use quantization for applications that are tolerant to overflows. We observe that in most applications, the rate of outliers is low and values are often within a narrow range, providing the opportunity to exploit quantization in general-purpose processors. However, a software implementation of quantization in general-purpose processors has three problems. First, the programmer has to manually implement conversions and the additional instructions that quantize and dequantize values, imposing a programmer's effort and performance overhead. Second, to cover outliers, the bitwidth of the

quantized values often become greater than or equal to the original values. Third, the programmer has to use standard bitwidth; otherwise, extracting non-standard bitwidth (i.e., 1-7, 9-15, and 17-31) for representing narrow integers exacerbates the overhead of software-based quantization. The key idea of this chapter is to propose hardware support in the memory hierarchy of general-purpose processors for quantization, which represents values by few and flexible number of bits and stores outliers in their original format in a separate space, preventing any overflow. We minimize metadata and the overhead of locating quantized values using a software-hardware interaction that transfers quantization parameters and data layout to hardware. As a result, our approach has three advantages over cache compression techniques: (i) less metadata, (ii) higher compression ratio for floating-point values and cache blocks with multiple data types, and (iii) lower overhead for locating the compressed blocks. It delivers on average 1.40/1.45/1.56× speedup and 24/26/30% energy reduction compared to a baseline that uses full-length variables in a 4/8/16-core system. Our approach also provides 1.23× speedup, in a 4-core system, compared to the state of the art cache compression techniques and adds only 0.25% area overhead to the baseline processor.

1.1.2 Fulcrum: a Simplified Control and Access Mechanism toward Flexible and Practical In-situ Accelerators

In-situ approaches process data very close to the memory cells, in the row buffer of each subarray. This minimizes data movement costs and affords parallelism across subarrays. However, current in-situ approaches are limited to only row-wide bitwise (or few-bit) operations applied uniformly across the row buffer. They impose a significant overhead of multiple row activations for emulating 32-bit addition and multiplications using bitwise operations and cannot support operations with data dependencies or based on predicates. Moreover, with current peripheral logic, communication among subarrays is inefficient, and with typical data layouts, bits in a word are not physically adjacent.

The key insight of this chapter is that in-situ, single-word ALUs outperform in-situ, parallel, row-wide, bitwise ALUs by reducing the number of row activations and enabling new operations and optimizations. Our proposed light-weight access and control mechanism, Fulcrum [13], sequentially feeds data into the single-word ALU and enables operations with data dependencies and operations based on a predicate [14]. For algorithms that require communication among subarrays, we augment the peripheral logic with broadcasting capabilities and a previously-proposed method for low-cost inter-subarray data movement. The sequential processor also enables overlapping of broadcasting and computation, and reuniting bits that are physically adjacent. In order to realize true subarray-level parallelism, we introduce a light-weight column-selection

mechanism through shifting one-hot encoded values. This technique enables independent column selection in each subarray.

1.1.3 Gearbox: A Case for Supporting Accumulation Dispatching and Hybrid Partitioning in PIM-based Accelerators

Recent PIM works employ processing units with a SIMD architecture to provide high parallelism.

However, kernels with random accesses cannot effectively exploit the parallelism of these approaches. Without efficient support for random accesses, ALUs remain idle until all the operands are collected from local memory segments (memory segment attached to the processing unit) or remote memory segments (other segments of the memory).

Generalized sparse-matrix-dense-vector (SpMV) and sparse-matrix-sparse-vector (SpMSpV), used in a wide range of applications, require random accesses. For SpMV and SpMSpV, properly partitioning the matrix and the vector among the memory segments is also very important. Partitioning affects (i) how much processing load will be assigned to each processing unit and (ii) how much communication is required among the processing units. In SpMSpV, unlike SpMV, the load assigned to each processing unit depends on the non-zero entries of the input vector, making partitioning even more challenging.

In this chapter, first, we propose a highly parallel architecture that can exploit the available parallelism even in the presence of random accesses. Second, we observed that, in SpMV and SpMSpV, most of the remote accesses become remote accumulations with the right choice of algorithm and partitioning. The remote accumulations could be offloaded to be performed by processing units next to the destination memory segments, eliminating idle time due to remote accesses. Accordingly, we introduce a dispatching mechanism for remote accumulation offloading. Third, we propose *Hybrid partitioning* and associated hardware support. Our partitioning technique enables (i) replacing remote read accesses with broadcasting (for only a small portion of data that will be read by all processing units), (ii) reducing the number of remote accumulations, and (iii) balancing the load.

Our proposed method, Gearbox [15], with just *one* memory stack, delivers on average (up to) 15.73× (52×) speedup over a server-class GPU, NVIDIA P100, with *three* stacks of HBM2 memory.

1.1.4 Pulley: An Algorithm/Hardware Co-optimization for Multi-device In-memory Sorting

Processing-in-memory (PIM) minimizes data movement overheads and provides high parallelism by placing one/few ALUs near each memory segment (e.g., bank or subarray). Current PIM approaches are inefficient

for three types of kernels: (i) kernels that require significant random accesses, which are costly in PIM, (ii) kernels that require a large intermediate memory per ALU, because the capacity overhead is multiplied by the number of ALUs, and (iii) kernels that require an operation across all the large intermediate arrays, because such operations in PIM are often performed through a core far from memory segments.

Parallel sorting is an important and widely used kernel with these requirements. In this paper, we propose an algorithm and hardware co-optimization for sorting that reduces the capacity and performance overhead of intermediate arrays and eliminates random accesses. To this end, we add hardware support that enables every group of processing elements to share an intermediate array, reducing the number of required arrays. We also modified the sorting algorithm to eliminate random accesses. Our hardware/algorithm optimizations, Dubbed Pulley, on average, delivers 20× speedup compared to Bonsai [16], an FPGA-based sorting accelerator and 13× speedup compared to IMC [17], an in-logic-layer-based sorting accelerator.

1.2 Overview of Dissertation

Our evaluations of our proposed methods shows that light-weight compression and flexible near-data processing can significantly mitigate the cost of data movement. Our light-weight compression provides 1.23× speed up compared to the state-of-the-art cache compression techniques and can significantly reduce the cost of data movement, supporting the first part of our hypothesis. Fulcrum [13], our near-data processing approach, provides 70× speedup per memory stack and reduces the energy consumption by 96%, significantly reducing the cost of data movement and supporting the second part of our hypothesis. Our second version of our near-data accelerator outperforms Gunrock, a GPU-based graph-processing framework, by 15.73×, confirming that near-data processing can significantly reduce the cost of data movement for graph processing applications, supporting the second part of our hypothesis. The third version of our near-data accelerator, on average, delivers 20 × speedup compared to Bonsai, an FPGA-based sorting accelerator, supporting the second part of our hypothesis.

The remainder of this dissertation is organized as follows:

Chapter 2: Background and Prior Works introduces the problem of data movement, memory hierarchy, and prior works.

Chapter 3: An Overflow-free Quantized Memory Hierarchy in General-Purpose Processors presents a light-weight, overflow-free, and narrow-bitwidth value representation in the memory hierarchy of general-purpose processors. This chapter, in full, is a reprint of the material as it appears in the proceeding of IISWC 2019 [18]. I was the primary investigator and author of this paper.

Chapter 4: Fulcrum: a Simplified Control and Access Mechanism toward Flexible and Practical In-situ Accelerators presents a flexible but practical subarray-level processing approach. This chapter, in full, is a reprint of the material as it appears in the proceeding of HPCA 2020 [13]. I was the primary investigator and author of this paper.

Chapter 5: Gearbox: A Case for Supporting Accumulation Dispatching and Hybrid Partitioning in PIM-based Accelerators presents how we extend Fulcum for highly sparse operations. This chapter, in full, is a reprint of the material that is accepted to appear in ISCA 2022 [15]. I was the primary investigator and author of this paper.

Chapter 6: Pulley: An Algorithm/Hardware Co-optimization for Multi-device In-memory Sorting presents how we extend Fulcrum for large-scale sorting. I am the primary investigator of this work.

Chapter 8: Conclusions summarizes the dissertation and discusses the implications of this work and potential future directions of research.

Chapter 2

Background and Related works

2.1 Background

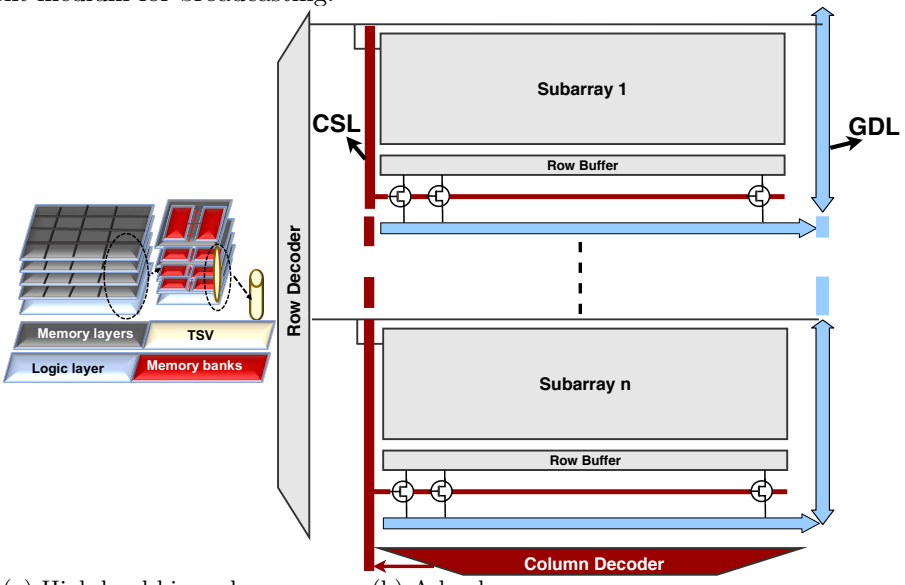
In this section, we introduce the structure of 3D stacked DRAM memories and their specific components that we have employed in our proposed method.

In 3D stacked DRAM memories, unlike DIMM-based memories, data is not interleaved among several chips. This characteristic is the first characteristic that makes 3D stacked memories suitable for PIM-based accelerators because processing inside memory is most efficient when all the bits of values are physically adjacent.

3D stacked DRAM is organized in a multi-level hierarchy, as illustrated in Figure 2.1 (a). Each 3D stacked memory has one logic layer and multiple (e.g., 4 or 8) memory layers. Each stack is divided into multiple vertical partition [19]] (e.g., 16 or 32), vaults/channels. Each with its own memory controller in the logic layer, connected to DRAM partitions. Layers are connected via TSVs (through-silicon via). The TSV is shared among all the banks in the vault. This shared bus is a bottleneck, as it serializes access to all subarrays in a vault. It is possible to employ a segmented TSV [20, 21], where every two layers are connected through one separate TSV, and data from upper layers are buffered in each layer before being sent to the lower layers. Although segmented TSVs increase the access bandwidth (because it acts as a pipeline, where the latency of each stage is lower than the latency of one-segment TSV), the segmented TSV increase the latency and the energy consumption of data movement (due to the extra cost of buffering and arbitration) [20, 21].

In each layer of each vault, there are a few banks (e.g., 2 or 4). Figure 2.1 (b) shows the structure of a memory bank. A bank comprises several subarrays. Each subarray has multiple rows. Each row contains multiple columns of data. The column width varies in different DRAM configurations. The most common

column widths are 32, 64, 128, and 256 bits. To access one column of the data from a bank, a row address is sent to all row decoders in all subarrays through the address bus in the bank. The row decoder selects the corresponding subarray and the corresponding row. The whole row that contains multiple columns is read out at once and will be stored in a row-wide buffer in the subarray, which is called the row buffer. To select one column from the row buffer, the column address should be sent to the column decoder, at the edge of the bank. The column decoder decodes the column address and sends the decoded bits to each subarray through column selection lines (CSL). The pass transistors/multiplexers in each subarray receive the decoded column address on CSLs and select the requested column and send it to the local data line (LDL), in each subarray. The data on LDL is sent to the logic-layer via global data lines (GDL). Two components make 3D stacked memories more suitable for Fulcrum: (i) the logic layer, and (ii) shared TSVs. The logic layer is more efficient for implementing logic than memory layers. Therefore, we can place flexible core and SRAM memories in the logic layer and exploit these components for parts of the applications that are not suitable for in-memory processing. We also employ the shared TSVs for broadcasting, because TSVs are shared among all banks in a vault and provide an efficient medium for broadcasting.



(a) High-level hierarchy (b) A bank

Figure 2.1: 3D DRAM: (a) each 3D stack has a logic-layer and a few memory layers containing multiple memory banks, (b) each bank comprises several subarrays

2.2 Related Work

This section introduces several categories of prior works on reducing the cost of data movement.

2.2.1 Bit-width Reduction Techniques.

Several studies employed the OLSB method for ASIC and FPGA designs to customize the *bit-width* [8, 9]. Prior works also show the energy reduction of quantization with arbitrary *bit-width* in accelereter design and provide a framework for deciding the *bit-width* [10, 11]. However, these solutions cannot be adopted in the general-purpose processors, because the bitwidth of variable in general-purpose processors are limited to only a few standard options (e.g., 8, 16, and 32). The only general-purpose solution, ACME [1] employs the OLSB method which has lower accuracy and lower performance than quantization. We evaluate both our proposed method and ACME with the specific bit-width that achieves the same level of average relative error (ARE) and maximum absolute error (MAE). Our evaluations show that our proposed method can achieve up to 31%/26% performance improvement with ARE/MAE fixed to 5%. Our approach provide the same accuracy with shorter variables and therefore, it achieve higher speedup.

2.2.2 In-situ Computing with Bitwise Operations

SRAM-based [22] and DRAM-based [23, 2, 12, 24] in-situ accelerators often support only bitwise operations. An NVM-based in-situ accelerator, Pinatubo [25], also proposes to change the sense amplifiers to enable bitwise operation for NVMs. A limited number of applications can benefit from bitwise operation. Recent works [2, 26] implemented binarized and 2-bit quantized deep neural networks using in-situ accelerators. In most memory technologies, employing row-wide complex ALUs imposes a significant hardware overhead. Li et al., [2] evaluated the overhead of adding a 4-bit adder per every four bits of the row buffer and concluded that it imposes more than 100% area overhead. Our proposed method has two advantages over these works. First, we enable 32-bit complex operations such as multiplication and addition operations without imposing a substantial hardware overhead. Second, our proposed method increases the flexibility of accelerators by supporting conditional operations, operations with data/position dependency, and independent random accesses to support a wider range of applications.

2.2.3 In-situ NVM-based Computing

Several prior works [27, 28, 29] employed the analog computation capability of ReRAM technology to perform matrix-vector multiplication. Our proposed method differs from such methods in three aspects. First, these technologies are memory-technology dependent and often use multi-bit memristor devices, which are unreliable. These techniques are neither applicable to SRAM/DRAM, nor commercialized NVM memories [30] such as 3DXpoint [31]. Second, they require analog-to-digital converter (ADC) and digital-to-analog converter (DAC) (ADC/DAC) blocks that impose significant hardware overhead (98% of the total area) and power overhead

(89% of the total power consumption) [30]). Third, performing multiplication and addition operations by approximately measuring the current introduces potential imprecision. A recent work, FloatPIM [30], designs a CNN accelerator for training by enabling floating-point matrix-vector multiplication in memory blocks, without requiring ADC/DAC. To this end, this approach copies the vector (which is the shared value), not only in each subarray but per each row of the matrix, to enable parallel multiplication and addition, imposing capacity and energy overhead. They change the entire memory architecture and interconnection, and as a result, the memory cannot be efficiently be accessed as a normal memory. Furthermore, it implements addition and multiplication using multiple bitwise operations and depends on the computation capability of memristors. Furthermore, it implements addition and multiplication using multiple bitwise operations and depends on the computation capability of NV memories. Since NV memories allow a limited number of write operations, NVM-based solutions that frequently write to memory have low endurance. For example, GRAM blocks fail after only one day. Therefore GRAM's authors propose to employ endurance management techniques that extend the lifetime to 496 days [32].

2.2.4 Flexible Cores in DRAM Layers

In the first round of research on processing in memory, in the 1990s, a variety of works [33, 34, 35, 36, 37, 38]) proposed to add flexible cores per each bank or per entire DRAM chip. Some of these proposals [39] add one processor per entire DRAM, plus multiple bank-level buffers. In these proposals, each column of these buffers moves toward the single processor. This imposes a high cost of data movement, is not scalable for modern DRAMs with many banks and subarrays, and limits parallelism. Our buffers enable sequential word-level access with subarray-level parallelism. Recently UPMEM [40] has described a product with a complex core per each bank of 64 MB. Fulcrum instead adds a simple processing unit per 1 MB and consequently provides higher parallelism and imposes less overhead for data movement (Our evaluations show that the energy consumption of accessing data at the edge of a bank through GDLs is at least 3× as much as that of a floating-point addition). More importantly, traditional cores impose a significant overhead of control and access for sequential operations. This overhead might be acceptable for far-memory cores for two reasons. First, for these cores, the ratio of this overhead compared to the overhead of data movement is negligible. Second, these cores use the small technology size and have a high frequency, whereas cores in DRAM layers have to use large technology size and have a low frequency. Consequently, the overhead of the extra cycles for such control and access mechanism becomes significant for in-memory cores. Fulcrum accelerates sequential operations and reduces the overhead of access and control.

Chapter 3

An Overflow-free Quantized Memory Hierarchy in General-Purpose Processors

3.1 Introduction

Data transfer across the memory system and interconnect constitute a significant fraction of the total energy and performance in memory-intensive applications [41, 42, 43, 44, 45, 46]. Prior works [4] show that the energy cost of fetching a 32-bit word of data from off-chip DRAM is 6400× higher than an ADD operation. This trend worsens as the processor technology moves to smaller nodes.

Prefetching and forwarding techniques [47] can alleviate the performance cost but they can not reduce the energy cost of data movements. Therefore, several approaches proposed to trade off the accuracy for the size of transferred data by omitting (truncating) a certain number of least significant bits (LSBs) in the mantissa of floating-point numbers [1, 8]. We refer to these methods as OLSB. Two factors limit the benefit of such techniques: (i) the overhead of eight bits for the exponent, and (ii) the high error that grows with the value of the exponent. In other words, the larger the magnitude of the value, the higher the absolute error (the magnitude of error). Despite the unfavorable effects, the floating-point format and the exponent part are necessary for supporting a wide range of values. In fact, the floating-point format is popular among developers because it supports a wide range of values and decreases the probability of overflow during arithmetic operations. Due to this popularity, processor vendors added floating-point ALUs

as soon as there were enough transistors available on the chip.

Our characterization demonstrates that, in real applications, most of the data values lie within a limited range and only a small fraction of data values are located at the tail of the distribution where these values are relatively further from the average. We define these values as *outliers*. Based on this observation, we make a case for using a specific type of *quantization* (biased fixed-point), as a mean to reduce the bitwidth of the variables and reduce the cost of data movement. This type of quantization maps a range of values to a set of discrete indexes and therefore it requires few bits to represent indexes [4, 48, 49, 50].

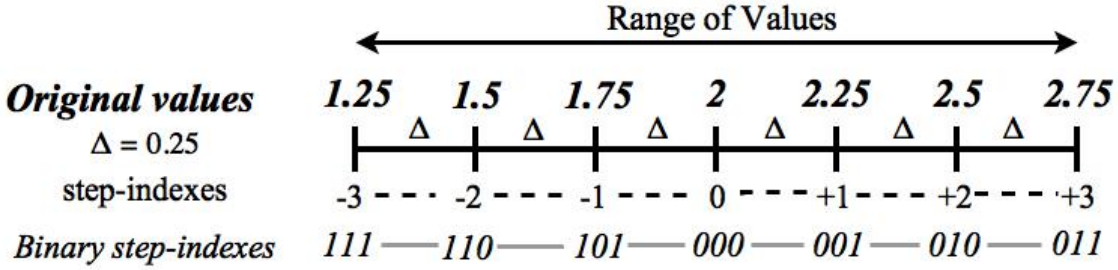


Figure 3.1: Quantizing a range of values to 3-bit integers

Figure 3.1 shows that this type of *quantization* reduces the number of bits required for representing values in the range of 1.25 to 2.75 by dividing the range into six steps (Δ=0.25) and mapping them into 3-bit integers (*step-index*). In this method, outliers can significantly expand the range and consequently increase the number of bits required to represent the quantized values, even though they are accessed very infrequently. A simple solution for handling outliers is to map values that result in overflows to the maximum or minimum of the range. The effect of such mapping depends on the application. For example, in BlackSholes, mapping outliers to the maximum or minimum of the range increases maximum absolute error by 116%, 333% and 69355% for bitwidth of 4, 8, and 12, respectively (as bitwidth increases our method's error decreases and hence the ratio of the error caused by outliers to our method's error increases).

A software implementation of quantization in general-purpose processors imposes a significant conversion overhead and programmer's effort, is prone to overflow, and more importantly, can not unlock maximum benefit for variables that require less bitwidth than standard variables. It has to use only 8-bit, 16-bit, or 32-bits variables and hence it imposes significant (e.g., 100%, 77%, or 88% for 4-bit, 9-bit, or 17-bit variables, respectively) cache space and memory bandwidth overhead (more details in Section 3.2).

Due to these constraints, quantization is popular in accelerators, where the bitwidth can be customized [9, 10, 11]. However, having one accelerator for each application, especially in consumer devices, is impossible, due to space constraints, scheduling overheads, communication overheads and, interconnection limitations.

The goal of this work is to propose and evaluate an overflow-free and transparent architectural support for quantization in memory hierarchy of general-purpose processors to accelerate a large domain of memory-

intensive applications, where variables can be represented with minimum and flexible number of bits. Our hardware modules act as accelerators for conversions that transparently quantize and dequantize cache blocks as they move between L1 and L2 (or alternatively between L2 and L3). Accordingly, the data values are quantized in the memory hierarchy in L2 and beyond and are de-quantized when transferred to L1, saving the capacity of L2, L3, and memory as well as bandwidth of L3 and memory. The values are in their original format in L1 and hence quantization causes no overflow during computation. To prevent overflow while we quantize and transfer data to L2, we propose to store and represent outliers in a separate space, assigned to outliers, and propose a mechanism for retrieving these values.

Quantization could be considered as a specific type of compression. However, cache compression techniques impose a significant metadata overhead and need a complex mechanism for locating the address of compressed values in compressed caches (translating the address). Due to these overheads, most of the cache compression techniques are only amenable for L3 [5, 6] and not applicable for L2 (more details in Section 3.2). We observed that the inefficiencies stem from the fact that compressor and decompressor have minimum information about the program and obliviously search for value locality within each cache block that they receive [5, 7]. We exploit the predictability of the range of values and fixed bitwidth in quantization and devised a software-hardware interaction to address inefficiencies in cache compression techniques. The interaction transfers specific characterization of applications such as data layout, distribution of values, and tolerable error (translated to *mid*, *step-size*, and *bit-width*) to hardware. Our hardware modules use this information to track which pages belong to which array of the application and hence store metadata only once for all pages of an array, reducing the metadata overhead. More importantly, our hardware modules exploit the bitwdith information for a light-weight address translation mechanism, implemented by arithmetic operations. This simplified address translation mechanism enables us to have a compressed L2 in addition to L3.

This paper makes the following contributions:

- We characterize 11 *real data sets* to show that real applications operate on data values within a particular narrow range, with only few outliers out of the range, suggesting that a significant portion of values can be represented by few bits.
- We propose efficient techniques to provide support for *quantization in hardware*. First, we propose a simple software-hardware interface to specify quantized variables and the necessary parameters. Second, we introduce efficient hardware modules that transparently quantize and dequantize cache blocks between a upper-level cache and a lower level cache. Third, we propose an efficient way of supporting outliers.
- Our evaluation of approximate applications shows that quantization provides on average 39-98% better accuracy compared to the techniques that omit the LSBs. Quantization provides on average a speedup of

1.40/1.45/1.56× and energy reduction of 24/26/30% compared to a baseline that uses full-length variables in a 4/8/16-core system. We have synthesized the RTL implementation of our hardware modules [51] and the synthesize report shows that our method adds only 0.25% area overhead to the baseline processor.

3.2 Motivation

In this section, we explain the benefit of hardware-based quantization over three alternative approaches: (i) quantization in software, (ii) OLSB, and (iii) cache compression.

3.2.1 Quantization in Hardware versus Software

A software-based implementation of the quantization method suffers from four disadvantages. First, it imposes the overhead of multiple instructions for each conversion. Figure 3.2 (a) demonstrates multiple examples of necessary conversion points: (i) quantization before storing values in arrays (❶ and ❸), (ii) dequantization before functions that requires the real values, such as sine and cosine (❷), (iii) dequantization before computation on non-quantized values (❹), (iv) conversion to avoid overflow (❺), and (v) dequantization before storing the final results in the output file (❻). Second, it is error-prone as programmers should manually detect the locations of necessary conversions. Third, when the required *bit-width* is 1-7, 9-15, 17-31, it uses 8-bit, 16-bit and 32-bit variables, respectively (to avoid the overhead of addressing and extracting a few bits in a sequence of bits), imposing up to 700%, 77% and 88% overhead of cache space and memory bandwidth. Fourth, it cannot represent outlier values, which is quite common in real data sets (explained in Section 3.3). For example, in Blackscholes, most price values can be represented by 6 bits but the maximum price requires 18 bits. With no support for outliers, software-based quantization has to use 32 bits for all values. We can implement our proposed method for outliers in software and store them in a separate array. However, in this case, quantization (❼) and dequantization (❽) functions becomes more complex as they have to check outliers and read/write outliers from/in a separate space(Figure 3.2 (b)).

3.2.2 Quantization versus Omitting the LSBs.

This section discusses two major benefits of using quantization in approximate applications.

***(i)* Lower relative error/shorter bit-width.** Figure 3.3 (a) compares the error introduced by quantization to the error introduced by OLSBs while varying the bit-width from 4 to 30 in eight popular approximate applications (details on the methodology is available in Section 3.6.1). For OLSB, we use one bit for the sign and the rest of the bits for mantissa (unless the bitwidth is 4 bits, where we use one bit for the sign and three bits for exponent and assume mantissa is one). The accuracy metric here is the *average relative*

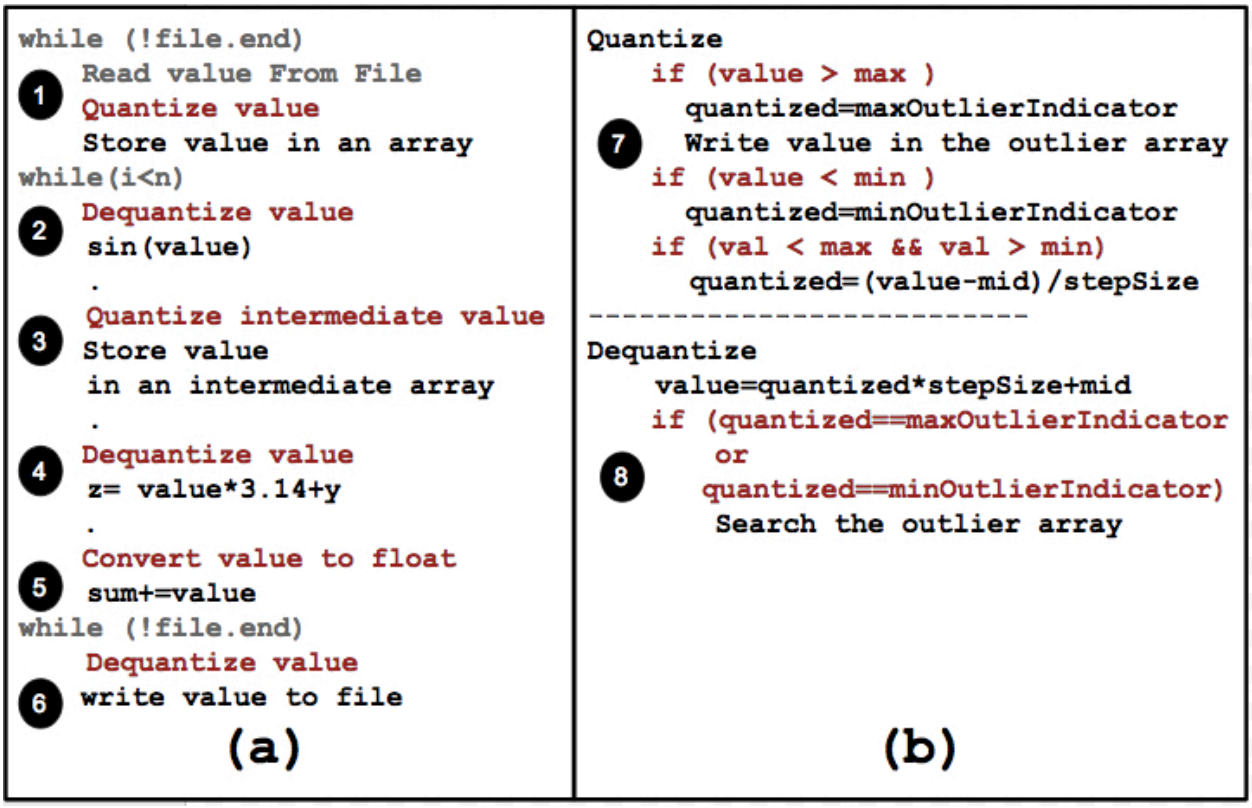
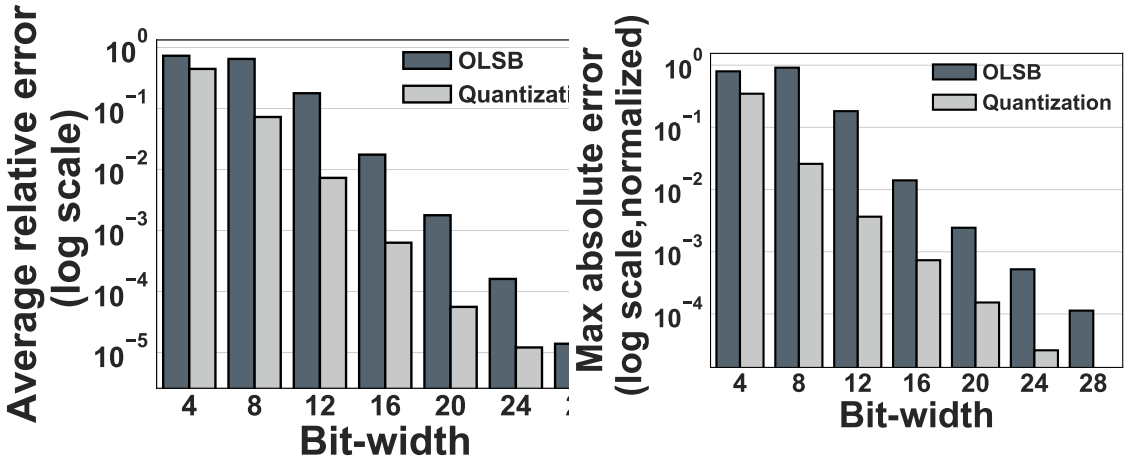


Figure 3.2: Quantization in software

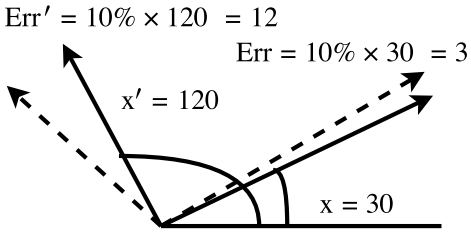


(a) Relative error

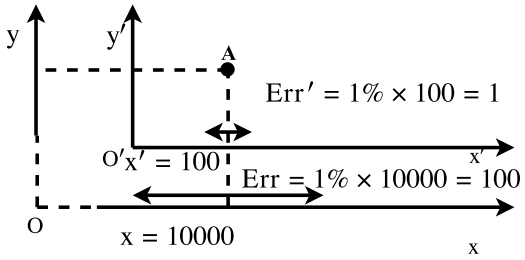
(b) Absolute error

Figure 3.3: Final output error using quantization vs. OLSB

error (average of the percentage of error for all output variables), which has been used in prior works on approximation [1, 58, 59]. This figure clearly shows that, for any given bit-width, quantization's error is lower than OLSB's error. It also shows that, for the same level of error, quantization requires fewer bits. Shorter bit-width stems from the fact that, unlike OLSB, quantization does no need eight bits for the exponent part.



(a) Angles



(b) Coordinates

Figure 3.4: Relative error for (a) angles, and (b) coordinates

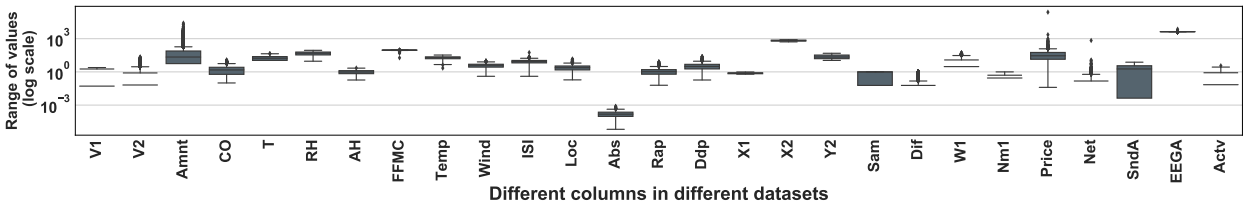
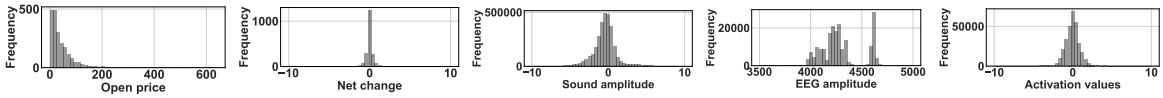


Figure 3.5: Variables in real datasets exhibiting a limited range (details in Table 3.1)

***(ii)* Lower absolute error/shorter bit-width.** When the accuracy metric is the relative error, the magnitude of the error can grow with the magnitude of the original value. However, many approximate applications expect that the magnitude of error remains limited regardless of the original value. For example, Inversek2j (an application from AxBench suit [58]), is an application that calculates the rotation angle for a 2-joint robotic arm. Assume that we define a relative error of 10% as the tolerable error. In this case, if the arm moves a small angle to hold the object, such as 30°, it will be off by only 3°, but for large angles, such as 120°, the error becomes 12°, which is quite high and can potentially make the arm miss the target object (Figure 3.4(a)). In reality, the acceptable error depends on the diameter of the target object, which is a fixed value and does not depend on the original value of the rotation. Another example is the inputs of Jmeint (from AxBench suit [58]) that analyzes the overlap of a pair of triangles in the 3-D space. In the real world, the acceptable error for the coordinates of the point A should not depend on the location of the center of the cartesian system (Figure 3.4(b)). For these applications, the absolute difference between the original value and the approximate value defines the proper accuracy metric.



(a) Blksh, NYSE [52](b) Financial, NYSE [52] (c) FFT, ASD [53, 54]) (d) FFT, EEG [55] (e) VGG, Cifar10 [56, 57])

Figure 3.6: Histograms of data values in various applications

Unfortunately, when we omit the LSBs from the mantissa, the absolute error depends on the value of the exponent ($Error = (-1)^S \times \left(\sum_{i=23-(l+1)}^{23} M_{23-i}2^{-i}\right) \times 2^E$) which can lead to a high absolute error if the exponent value is large whereas quantization limits the maximum possible absolute error to the *step-size*. Figure 3.3 (b) demonstrates that, compared to OLSB, quantization lowers maximum absolute error in the output of our evaluated approximate applications when the *bit-width* is varied from 4 to 30.

3.2.3 Quantization versus Cache Compression.

Cache compression techniques have three problems. First, they require metadata per cache block. For example, Base-delta [7] shrinks the size of each cache block by subtracting the values within the block from a base value. It requires 1-4 bytes for the base value per block. For a compression ratio as high as four, the four bytes, per compressed block, imposes 25% overhead. Second, they can not achieve a reasonable compression rate for two types of arrays: (i) arrays containing single-precision floating point values, where variation in the least significant bits is high and (ii) arrays of structure or any other composite data type with consecutive variables that are inherently different. Third, cache compression techniques, compress each block of the cache into different size which requires a complex mechanism for locating the cache blocks. As a result, they employ one of the three following techniques: (i) padding that pads compressed block so that the size of each compressed block becomes an integer and power of two fraction of the size of the original cache block (e.g. LCP [60]) (i) dividing the original cache block into integer number of segments and assigning one tag per each segment (e.g. HyComp [5]), (ii) employing a completely decoupled tag array and data array, where the tag array points to the start of the compressed block in the data array (e.g. Decoupled compressed cache [6]). The first and second approach constraint compression ratio and the third approach requires modification in the cache and a defragmentation mechanism for the data array. More importantly, tag and data can not be accessed in parallel and will be accessed sequentially, doubling the latency of caches (which is already around 36 cycles for modern large last level caches). Since the complex decoupling mechanism cannot be employed for L2 caches, most cache compression techniques only compress the last level cache. In quantization, the compressed bitwdith is fixed and the page offset of each compressed variable can be determined by arithmetic operations. Section 3.4.4 explains that we exploit the fixed bitwidth to

locate our compressed values using simple arithmetic operations and keep the original structure of the cache, eliminating the decoupling mechanism. This enables us to quantize values in L2 in addition to L3. More importantly, by transferring information about data layout to hardware, we track which pages belong to which array of the application and hence store metadata only once for all pages of an array.

3.3 Key Observations and Key Ideas

This section explains three key observations that form three key ideas of this paper.

Observation 1: Data Values in a narrow Range. We characterize 11 *real data sets* [61, 62, 63, 64, 65, 66, 52, 54, 55, 57] used in different domains, such as machine learning, weather forecast, financial analysis, signal processing, image recognition, etc. (details in Table 3.1). Figure 3.5 illustrates the box plot for several variables in these datasets, where the box shows the range of values in the first to third quartile, the bars show the lower and upper limit within 1.5× of the first and third quartile, and the values outside that range are shown as dots. This figure clearly demonstrates that data sets for many approximate applications exhibit values within a narrow range. The first key idea is that, given the narrow range of values, quantization can be applicable to a wide range of applications and efficiently reduce the cost of data movement.

Table 3.1: Description of real data sets

Variable name	Data set description
V1, V2, Amnt	Credit card fraud detection data set [61]
CO, T, RH, AH	Sensor data for air quality [62]
Temp, Wind, ISI	Weather index for forest fire [63]
Loc, Abs, Rap, Ddp	Speech data from parkinson patients [64]
X1, X2, Y2	Intrusion detection data set [65]
Sam, Dif	Building shapes for energy efficiency [66]
W1, Nm1	Purchase in sale transactions [67]
Price, Net	Stock exchanges [52]
SndA	Sound amplitudes [54]
EEGA	EEG amplitudes [55]
Actv (VGG)	Images for deep learning [57]

Observation 2: Outliers. Figure 3.5 also demonstrates that some values will be outside the limited range (outliers) and Figure 3.6 shows that data values in most real applications exhibit a normal/folded normal distribution. According to the normal distribution definition, 99.99% of the data values are within 8 standard deviation and only 0.0001 (0.01%) of the values fall outside this range. There are two common approaches to deal with outliers in data analysis techniques [68]: *(i)* mapping outliers to a minimum or maximum value, or *(ii)* processing the outliers with their original values if the outliers provide meaningful insight to the analysis [68, 69, 70, 71]. Our proposed method provides support for both approaches. The second key idea

is to use the lowest possible number of bits for the most common values and store and represent outliers separately, for applications that require support for the second approach.

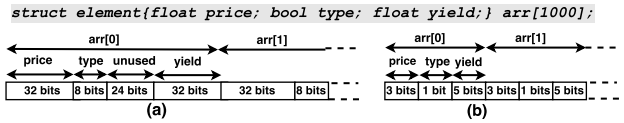


Figure 3.7: (a) Original and (b) quantized array of structure

Observation 3: Common data layout in memory intensive applications. Pointers impose a significant overhead [72, 73, 74]. Consequently, memory intensive applications, with high spatial locality, layout data in two different ways: (i) array of structures (AoS), or (ii) structure of arrays (SoA). For example, in graph processing applications (which intuitively should use a linked list), we prefer arrays of edges and vertices or sparse matrices[75], partly, because pointer chasing, dynamic memory allocation (for each element of the data structures such as linked list) and additional random memory access of linked lists imposes a significant overhead. Supporting quantized SoA is straightforward as consecutive elements have same quantization parameters. However, providing support for AoS requires a complex metadata handling and address translation mechanism (Figure 3.7 shows how AoS should be quantized). Accordingly, the third key idea is communicating data layout to hardware and track which pages belong to which arrays to keep metadata once per whole array and use the layout information for simplifying address translation. Communicating the layout, also enables eliminating the unused bits before byte variables (used for alignment of composite data types, as shown in Figure 3.7), compressing boolean values (which only need one bit), and compressing integers within narrow range (assuming the step-size is equal to one).

3.4 Mechanism

We quantize and dequantize value in the memory hierarchy and keep the ISA, pipeline, load-store module, controller, and data-path intact for three reasons. First, providing support for quantization in the processing unit calls for invasive modifications in cores. Second, previous studies show that employing short variables in the computation part of the systems, such as ALU, can significantly increase the error (due to arithmetic overflow and inaccurate intermediate values) [1, 76]. Third, the cost of moving data is 6400 times higher than ALU operations in modern processors [4]. Therefore, we focused on reducing the cost of data movement. The location of conversion can be decided based on different tradeoffs (performance vs. power). Hereafter, we assume that the conversion occurs between L1 and L2 caches, as it provides higher speedup by increasing the effective size of L2. However, we also evaluate the performance improvement when the conversion point is between L2 and L3 cache such that values are stored in the full-length format in both L2 and L1 caches

(Section 3.6.6). In this section we answer seven questions: (i) how to determine the quantization parameters?, (ii) how to transfer metadata to Hardware?, (iii) how to retrieve metadata?, (iv) how to locate quantized values?, (v) how to quantize and de-quantized values?, (vi) how to handle corner cases?, and (vii) how to avoid overflows?.

3.4.1 How to Determine the Quantization Parameters (Metadata)?

We observe that quantization parameters only depend on the nature of data and do not change significantly with different data sets. For example, many speech recognition applications process the amplitude of people's voice [53, 54], which does not drastically change across different datasets. In the modern development process, applications (such as machine learning applications) are trained and tested using some data set before the deployment. During the execution (inference) phase, the application process data with the same nature. Therefore, the quantization parameters can be derived using an offline profiler during the training and testing phase. To determine the effectiveness of the offline profiling, we divide our dataset into training and testing sets. Similar to machine learning training and testing, we kept at least 10% of the data for testing. For datasets such as NYSE [52], we tracked the stock prices for 20 days to have more than 80000 observations for training and ten days for testing. Some datasets such as CIFAR-10 [57] already have separated testing and training sets (There are 50000 training images and 10000 test images). We used their partitioning for testing and training. We profile applications using the training set to find the parameters for some specific average relative error (e.g., less than 10%/5%/1%) and find that the parameters provide similar accuracy even with the testing sets (the average relative error is 7%/2.2%/ 0.64%). Note that the quantization parameters can be selected conservatively with a small overhead. For example, a conservative selection of $0.5 \times$ *step-size* (the smaller the *step-size*, the lower the error) increases the bit-width only by *one bit*. Additionally, the offline profiling does not need to be 100% accurate as our mechanism is capable of handling a small fraction of outliers (explained in section 3.4.7).

Profiling can be a automated process. A parser detects arrays and the structure of arrays, then passes this information to the profiler. The profiler detects whether the arrays are large enough and extracts the quantization parameters for the specified tolerable error. A similar automated approach is being used for accelerator designs [11, 8].

3.4.2 How to Transfer Metadata to Hardware?

After profiling, we need to communicate the quantization parameters obtained by profiler to software and then from software to hardware. We have two options for transferring quantization parameters to software:

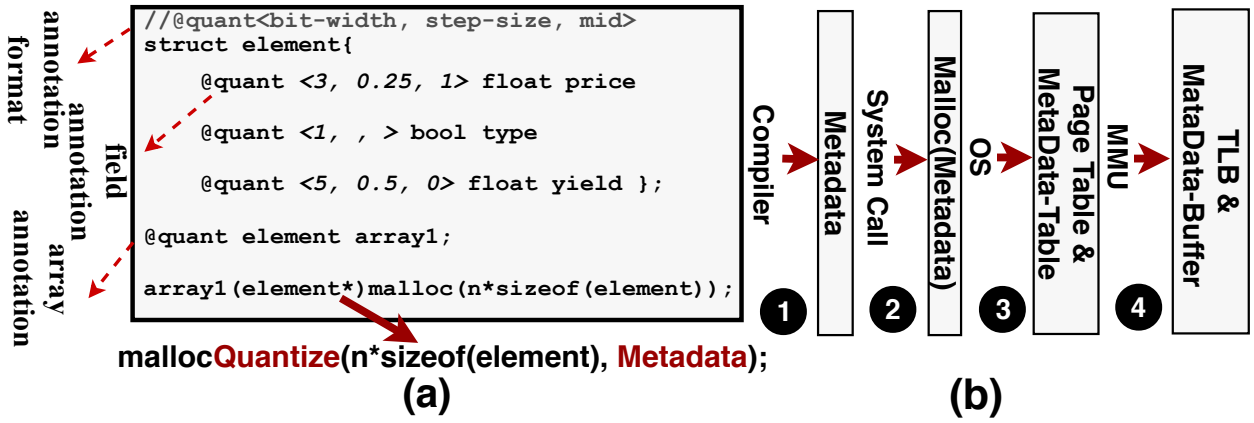


Figure 3.8: Software-hardware interaction

(i) automatic and static annotation of arrays with quantization parameters as shown in Figure 3.8 (a), and (ii) putting the parameters in input arguments, so that it can be read at run-time, dynamically (useful for API and library developers).

There are three essential steps involved in the communication of quantization parameters to hardware. First, the compiler extracts the metadata from the annotated code. It extracts *Bit-width*, *Step-size*, and *Mid* values of each filed of the structure (or address of the variables, in which these values are stored), as shown in Figure 3.8 (a) and Figure 3.8 (b), step ❶. (For this step, we are mimicking the compiler support and do not change the compiler itself). Then, the compiler passes this data to OS, using a specialized malloc function (system call in step ❷). Second, the malloc function assigns a page aligned space to the array and calculates other required metadata (explained in Section 3.4.3). At this point, OS stores a few fields of metadata required in the critical path in the page table (we extended page table entries to store these fields), and the rest of metadata, in our proposed table, `MetaData-Table` (step ❸). Third, once a page is requested, memory management unit (MMU), as a part of its normal process, transfers page table entries to TLB, meaning that metadata stored in the page table are transferred to TLB automatically. Our customized MMU also transfers the corresponding metadata, stored in `MetaData-Table`, to our hardware module, called `MetaData-Buffer` (step ❹).

3.4.3 How to Retrieve Metadata?

Our proposed method requires two types of metadata. The first type of metadata, such as *DeQWordCount*, is required for address translation between L1 and L2. *DeQWordCount* determines how many words (of L1 values) can fit in a quantized L2 block. *DeQWordCount* is an integer number (in Section 3.4.6 we explain why it should be an integer number). Access to *DeQWordCount* is in the critical path as it is required for

sending a miss request to lower level cache (L2). Hence, *DeQWordCount* is stored in the TLB so that it can be accessed when the processor accesses the TLB for the traditional virtual to physical address translation. Existing systems, such as Intel x86-64 systems [77] have up to 15 unused bits in their TLB entries. Our proposed method requires 18 bits [78]. Accordingly, we only add three bits to the original TLB entry and the total overhead per core is 216 bytes (in a system with 64-entry first level TLB and 512-entry second level TLB). The second type of metadata, including *Step-size*, *Bit-width*, and *Mid*, is required for data conversion, We introduce a new hardware module, the `MetaData-Buffer`, to store the metadata required for data conversion. Thanks to our `MetaData-Buffer`, our method can mix compressible and non-compressible data. In our `MetaData-Buffer` (a full description of each field and its purpose is available at our online documentation [78]), for each variable of the structure, we have a one-bit field, called "Conv?" that determines whether that field needs conversion or not.

3.4.4 How to Locate Quantized Values?

When compression/decompression happens between two level of caches, page number and page offset of the values in the decompressed cache are different from those of compressed cache. Unlike most cache compression techniques, we do not need decoupling the tag array and the data array for address translation. In fact, our fixed bitwidth enables address translation using arithmetic operations, which is performed by two modules: (i) BitLocationFinder, and (ii) `Offset-Divider`.

We built upon prior works [60, 79] that allocate smaller page within a 4KB page and add few bits in the page table entries that point to the start of the smaller page within 4KB pages. Therefore we only need to translate page offset. We explain this process using two examples.

Example 1, an array of float numbers (float Arr[1000]: In this example, each element can be quantized with 5 bits, and each cache line of the system has 64 bytes (512 bits). As a result, each L2 block accommodates 102 quantized words(*DeQWordCount*=512/5=102 (the division is implemented by multiplication by reciprocal and takes 2.5 cycles)). Step❶, an L1 eviction happens. Step❷: `Offset-Divider` simply divides the word number of the virtual offset(Vofset) by *DeQWordCount*, to determine the index of the L2 block that contains the missed L1 block($index = (Vofset >> 2)/102$). Since the index is ready, L1 can send the write-back request to L2. Step❸: while L2 is finding the L2 block containing the evicted block, the BitLocationFinder finds the exact location of the evicted L1 block within the L2 block. To this end, the BitLocationFinder multiplies the quantized bitwidth by the remainder of the last division ($(Vofsett >> 2\%102) * 5$).

Example 2, an AoS: In this case, BitLocationFinder needs two metadata: (i) *AccLen* and (ii) *StartWord*. *AccLen* is a field of our `MetaData-Buffer`. Per each variable of the structure, *AccLen* stores the accumulated

quantized length of preceding variables in the structure. This field optimizes BitLocationFinder to lower the complexity of address location calculation. *StartWord* is a field in the TLB extension that indicates with which word of the structure the page starts. BitLocationFinder first finds how many complete structures exist before the start of the L1 block and then multiply this number by the quantized size of the structure.

```
lengthOfCompleteStructures=(((Vofsett>>2)% DeQWordCount)/NumWordInStructure)*
    QntStructBits.
```

Then it uses *AccLen* and *StartWord* to calculate how many bits are required for any partial structure before the start of the L1 block. The remainder of the last division determines how many words exist in the partial structure.

```
remainder=(((Vofsett>>2)%DeQWordCount)% NumWordInStructure)
```

BitLocationFinder determines with which word of the structure, the partial structure ends

```
end=((Vofsett>>2)+StartWord)%NumWordInStructure
```

Then it calculates the number of bits in the partial structure and adds this to the lengthOfCompleteStructures.

```
if(end> remainder) {
     start=end-remainder
  BitLocationStart= AccLen[end]-AccLen[start]+lengthOfCompleteStructures;
}else{
start=NumWordInStructure+end-remainder
BitLocationStart= AccLen[NumWordInStructure+1]-AccLen[start]+AccLen[end]+
    lengthOfCompleteStructures;
}
```

This operation, on cache misses, overlaps with reading the block from L2 and, on writebacks, lies out of the critical path and does not hurt performance (The above pseudo code is developed to simplify the explanation and differs from our optimized Verilog implementation).

3.4.5 How to Quantize and De-quantized Values?

The de-quantization process is on the critical path of a L1 miss whereas quantization only happens during the writebacks. Therefore, we design a parallel de-quantizer for efficient and fast conversion that performs the computation shown in Equation 3.1. Since the *Step-size* is a power of two number, the multiplication in Equation 3.1 can be implemented as a summation of the exponent of *Step-size* and exponent of $Index + QuantizedMid$ (In fact, we store the exponent of *Step-size* in the `MetaData-Table`) and a *leading one detector module* (LOD) can calculate the exponent of $Index + QuantizedMid$. We propose a 16-way parallel module where each of the ways takes a 32-bit word (extracted through a crossbar) from the cache block as its input

and converts them in parallel (Figure 3.9). We synthesize the modules using an industry-standard 1xFinFET, and the latency of the whole process is 1.6 ns (4.8 cycles in 3GHz frequency).

$$\tilde{X} = (Index + QuantizedMid) \times step - size \tag{3.1}$$

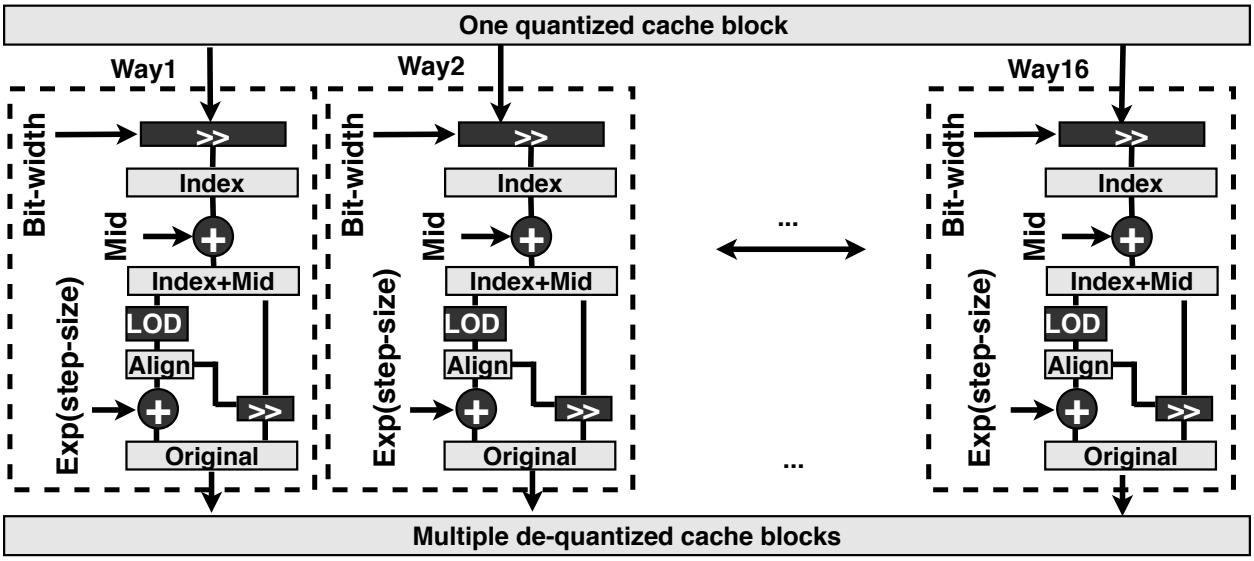


Figure 3.9: The De-quantizer module

3.4.6 How to Handle Corner Cases?

As an extra optimization, we relax the constraint of fitting only integer number of L1 blocks in L2 blocks to avoid the overhead of padding. As a result, naively unpacking a L2 block generates one or two partial L1 blocks on de-quantization, wasting cache space and requiring some extra metadata per cache block to track the valid words of the block. Alternatively, we place the partial blocks in the prefetch buffer until the rest of the block arrives to avoid extra metadata and wasted space in L1. We evaluate a 4-entry prefetch buffer (similar to the original prefetch-buffer paper [80]). As our method is word-aligned, we only add one valid bit per word to the prefetch buffer (total overhead of 64 bits). On a L1 miss, our `Offset-Divider` generates a miss for the L2 cache block that contains the missed address. Subsequent requests to the valid part of the partial block are serviced from the prefetch buffer and cause no miss. Once an address from the rest of the partial block is requested, the `Offset-Divider` module will generate a miss request for the subsequent L2 block. Note that we always store an integer number of quantized words in an L2 block (determined by *DeQWordCount*). Therefore, No L1 word spans two L2 blocks, ensuring that `Offset-Divider` never generates two L2 miss for one word, avoiding complexity in miss status history register (MSHR).

3.4.7 How to Avoid Overflows?

We provide two options for dealing with the outliers that does not fit in short bitwidth and may cause overflows: *(i)* mapping the outliers to the maximum or minimum of the range that can be represented using the *same* bit-width used in the quantization process, and *(ii)* storing the outliers in the original floating point format in a separate space. We explain how we support the second option using a walk-through example that shows a scenario with an outlier in a block.

Step ❶: L1 evicts a cache block. Step ❷: the converter tries to quantize the evicted block and send it to the L2 but it detects an oulier in the evicted block. Step ❸: the converter fills the location of the value in the L2 block with an outlier indicator (we assign both all-one or positive-all-one values as outlier indicators. For example, for a quantized value with bit-width of four, outliers will be stored as "1111"). Therefore, the layout of data in L2 does not change. Step ❹: to preserve the real value of the outlier, the converter also sends a message to memory, containing the real value of the outlier along with Voffset of the value to be stored in the reserved space, extra b blocks at the end of each page, after the quantized values (In case the reserved block space cannot accommodate a new outlier, the memory controller informs the OS to convert the quantized page to a normal page). Step ❺: when this block is requested again (on a L1 miss), the converter sees the indicator and detects that the related value is stored as an outlier. Thus, the converter sends a memory request to get the real value of the outlier from the reserved space. To this end, the memory controller reads the reserved blocks for outliers and scans these blocks until it finds the Voffset of the requested variable. The real value of the outlier is stored next to the Voffset. Therefore, while L2 stores the outlier indicator, with the same bitwidth of quantized values, L1 always stores the real value of the outlier and we never give up the wide range that floating point format can support.

In our evaluation with real data sets, we notice that the fraction of outliers is significantly low (less than 0.005%) and the overhead of retrieving the outliers does not hurt the performance (evaluated in Section 3.6.6).

3.5 More Walk-Through Examples

In this section, we explain the necessary steps during loading a new page, a L1 miss, and a L1 eviction. The steps are marked in Figure 3.10.

Steps during loading a new page: There are three steps involved in this process: (i) once processor loads a new page, our proposed boolean filed, *IsQuant* in the TLB entry indicates whether the page is quantized or not, (ii) if the page is quantized, our proposed *MetaDataID* field of the TLB entry specifies the Id of the corresponding `MetaData-Set` in `MetaData-Table` of the corresponding virtual space, (iii) when a TLB entry

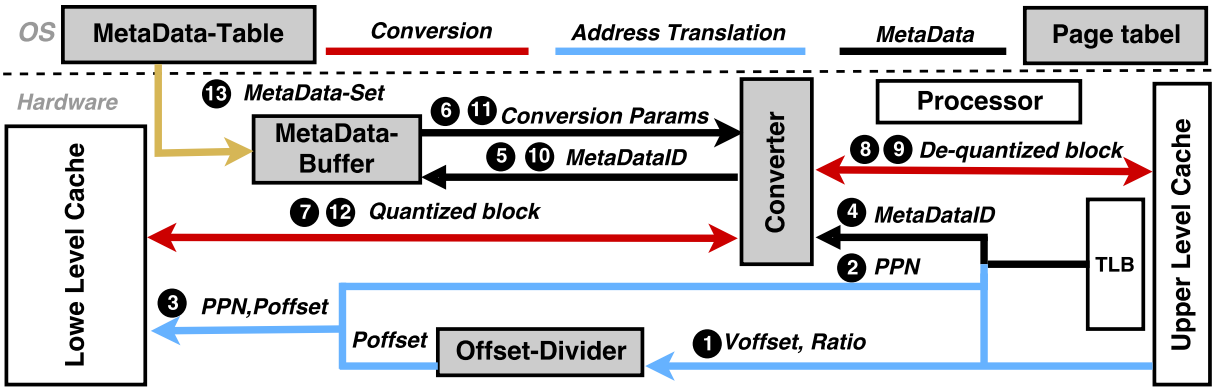


Figure 3.10: Overall architecture of hardware quantization

is loaded, the Address Space ID (ASID) + *MetaDataID* of the TLB entry is compared against the ASID+ *MetaDataID* of all the sets. In case of a miss, MMU assigns one of the free sets in the `MetaData-Buffer` to the *MetaDataID* and loads the corresponding `MetaData-Set` of the `MetaData-Table` in the `MetaData-Buffer` (Figure 3.10 ⓭). Note that each TLB entry traditionally has the ASID. Accordingly, we store ASID+ *MetaDataID* in each set of the `MetaData-Buffer`.

Steps during a L1 cache miss: There are three steps.

1. finding the address of the L2 block that contains the quantized values: Traditionally, for each load or store instruction, processor uses virtual offset (*Voffset*) to access L1 and virtual page number (*VPN*) to read the corresponding TLB entry and extract the the physical page number (*PPN*). In our method, on a L1 miss, if the *IsQuant* of the TLB entry indicates that the missed address belongs to a quantized page, L1 sends the *Voffset* to the `Offset-Divider` (Figure 3.10 ❶) and send *PPN* to L2 (❷). The translator divides the *Voffset* by the *DeQWordCount* value to generate the page offset (*Poffset*) of the L2 block and sends this offset to the L2 cache (❸).

2. Getting quantization parameters: While, waiting for the L2 block to be received, the TLB sends the *MetaDataID* to the `Converter` (❹). The `Converter` uses the *MetaDataID* to read the quantization parameters from the `MetaData-Buffer` (❺ and ❻).

3. Conversion: When L2 sends the quantized block to the `Converter` (❼), the `Converter` uses the quantization parameter to de-quantize the block and sends back multiple de-quantized blocks to L1 (❽).

Steps during a L1 write-back: There is no TLB access during the write-back. Accordingly, we store the *MetaDataID*, in the L1 cache tag. Quantization in write-back has two steps.

1. Getting quantization parameters: When L1 evicts a dirty block from a quantized page (❾), the *MetaDataID* is read from its tag and the `Converter` uses this *MetaDataID* value to read the quantization parameters from the `MetaData-Buffer` (❿ and ⓫).

2. Conversion: The `Converter` quantizes the block and calculates the location of the block in the quantized L2 block (using the BitLocationFinder module), and sends the quantized block to L2 (⓬). Upon receiving the block, L2 updates data at the correct location within the quantized block.

Table 3.2: The configuration of simulated systems

Component	Parameters
Processor	4/8/16 cores, 3 GHz, 8-wide issue
L1	64B cache-line, 2-way associative, 32KB, private, 4 cycles
L2	8-way associative, 256KB, private, 12 cycles
L3	16-way associative, 8/16/32 MB, shared, 36/43/58 cycles
Prefetch buffer	4-entry
Memory system	DDR3_1600_x64, 2 channels

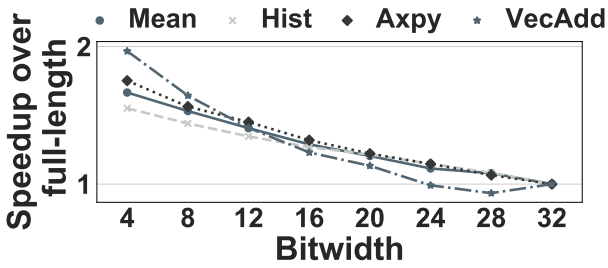
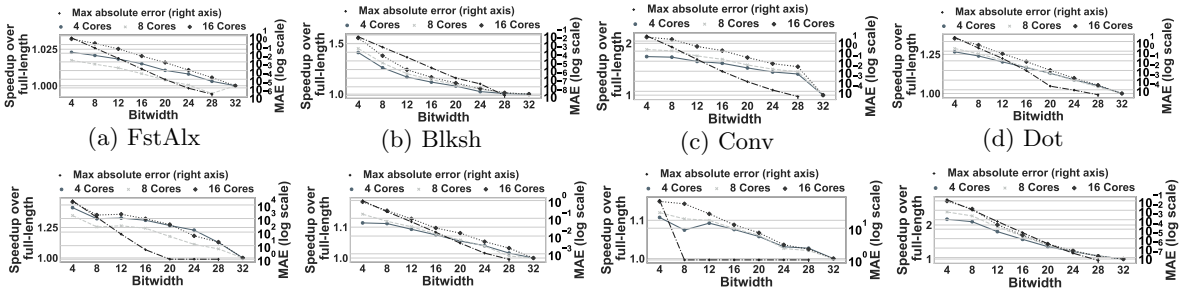


Figure 3.11: Effect of bitwidth on speedup for microkernels.

3.6 Evaluation

3.6.1 Methodology

To evaluate the performance and energy consumption of our proposed method, we augmented Gem5 [81] and McPAT [82] with timing and energy model of our proposed components. For simulation configuration, we use the specification of Intel Core-i7 Haswell processor (listed in Table 5.1) and add a penalty of five cycles (explained in Section 3.4.5) to model the conversion latency of a parallel converter module. We evaluated four microkernels: (i) Hist (calculates the histogram of a vector of values), (ii) Mean (calculates the mean of a vector), (iii) Axpy ($Y = \alpha \times X + Y$), and (iv) VecAdd ($Z = X + Y$). We also evaluated eight applications from different domains: four applications from AxBench [58] (Blksh, Sbl, Inv2j, and FFT), three convolution with configuration of three different layers of the deep learning application with high, moderate, and low locality (FstAlx, Conv, and LstVg), and a Dot product kernel (Dot). For the eight applications, we used standard data sets for each application [53, 54, 55, 83, 56, 57, 52]. Our evaluated workloads represent modern applications, which are highly memory-intensive and exhibits a large memory footprint. Prior works have shown that these applications are bottlenecked by the cost of data movement [4, 43, 84]. However, in order



(a) FstAlx (b) Blksh (c) Conv (d) Dot

(e) FFT (f) Inv2j (g) Sbl (h) LstVg

Figure 3.12: Effect of bitwidth on speedup and maximum absolute error

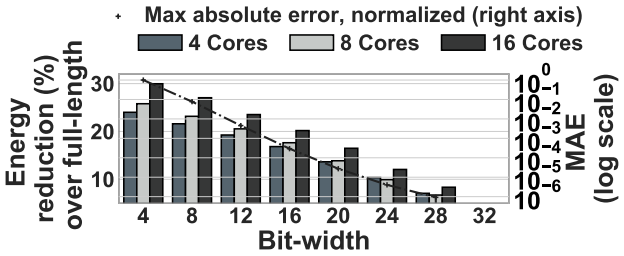


Figure 3.13: Energy reduction

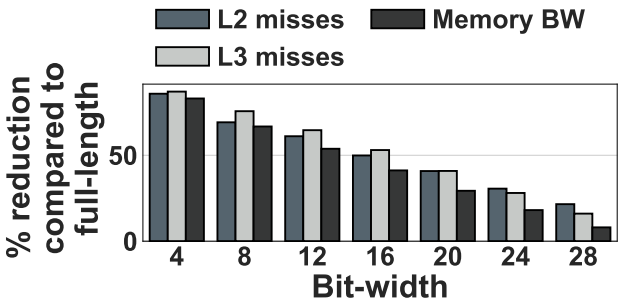
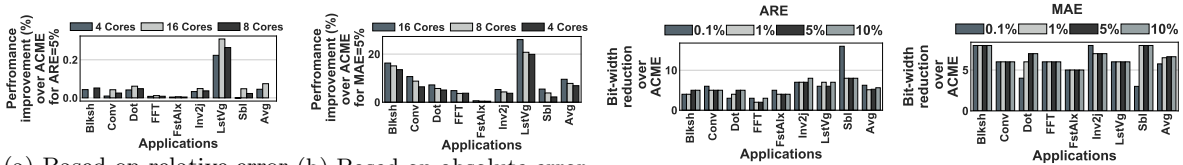


Figure 3.14: Source of improvement



(a) Based on relative error (b) Based on absolute error (c) Based on relative error (d) Based on absolute error

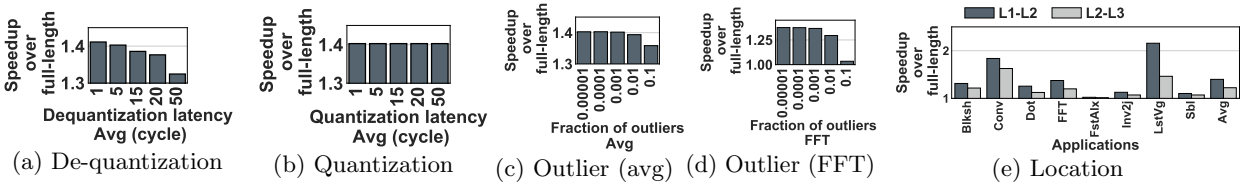
Figure 3.15: Comparison against ACME [1]

to be comprehensive, we also evaluated two computation-intensive applications (FstAlx and Inv2j), which inherently do not benefit from reducing the cost of data movement.

3.6.2 Performance Improvement and Error Analysis

Figure 3.11 illustrates the effect of bitwidth on speedup for four microkernels. Mean has one memory access per compouation, Hist has two memory accesses but one of them is often cached, Axpy has two memory accesses per computation and VecAdd has three memory accesses per computation. This figure shows that the narrower the bitwidth, the higher the speedup. It also shows that for VecAdd with three memory access per computation, a small reduction in bitwidth (e.g. bitwidth of 24) can not compensate the conversion penalty. However, for smaller bitwidth, such as four, the reduced memory bandwidth provide a significant speedup. Figure 3.12 shows the speedup (on the left axis) and maximum absolute error (on the right axis) per bitwidth for eight applications using real data sets. Our proposed mechanism achieves on average 1.6× (up to 2.7×) speedup over the baseline system.

There are four observations that we draw from Figure 3.12. First, the shorter the *bit-width*, the higher the speedup, demonstrating that reduction in data movement directly impacts the overall performance. In rare cases, such as Sobel, access pattern with the 8-bit variables causes more conflict misses than 12-bit variable, causing unexpected slowdown in a 4-core system. Second, a system with a higher number of cores gains higher speedup from reduction in data movement, due to the higher contention in caches and memory. Third, applications with a low temporal locality, such as LstVg, gains the most benefit from our proposed method. On the contrary, applications that have a high temporal locality gain the least speedup (e.g., FstAlx [85] achieves less than 1.04× speedup). Fourth, the shorter the *bit-width*, the higher the maximum absolute error, except for applications such as Sbl, where the values (pixel values) inherently can be represented by 8 quantized bits, without significant accuracy loss.



(a) De-quantization (b) Quantization (c) Outlier (avg) (d) Outlier (FFT) (e) Location

Figure 3.16: Sensitivity analysis

3.6.3 Energy Consumption

We model the energy consumption of all memory elements (such as `MetaData-Table`, etc.) and all arithmetic operations using McPAT [82]. Figure 4.13 demonstrates the average energy reduction of the evaluated benchmarks over the baseline when we vary *bit-width* and number of cores. It shows that the energy cost of data movement decreases when the applications use shorter *bit-width*.

3.6.4 Source of Improvement

The performance and energy benefits of quantization stem from two sources: *(i)* reducing the number of cache misses (due to an increase in the effective size of L2 and L3 and also due to loading one quantized block instead of multiple full-length blocks), and *(ii)* decreasing the memory bandwidth consumption. Figure 3.14 demonstrates that reducing the variable length from 32 bits to 6 bits reduces L2 misses/L3 misses/BW consumption on average by 77%/82%/74% in a 16-core system. None of the evaluated application fit in L1 and L2. Since the L3 is shared among cores, LstVg does not fit even in its share of L3 (before quantization). Some applications such as Blksh, fit in L3, but they do not show locality with long reuse distance, so they experience a high L3 miss rate. As an example, for Blksh, the L1 miss rate is 0.006052, the L2 miss rate is 0.877949, and the L3 miss rate is 0.999938. For FFT, L1 miss rate is 0.039064, L2 miss rate is 0.966428, and L3 miss rate is 0.34570956478. for LstVg, the L1 miss rate is 0.006861, the L2 miss rate is 0.917308 and 0.9999.

3.6.5 Comparison with Prior Works

Figure 3.15 compares the performance of our proposed method against ACME [1] that omits the LSBs to shorten the variable lengths. We evaluate both of the mechanisms with the specific *bit-width* that achieves the same level of average relative error (ARE) and maximum absolute error (MAE). Figure 3.15 (a) and 3.15 (b) show that our proposed method can achieve up to 31%/26% performance improvement with ARE/MAE fixed to 5%. We provide the same accuracy with shorter variables and therefore, achieve higher speedup. Figure 3.15 (c) and 3.15 (d) show the reduction in the number of bits compared to ACME when we vary the accuracy. On average, the bit width reduction in the quantized variables are 5.62/5.25/5.1/6.2 bits (ARE) and 6.62/6.62/6.5/5.7 bits (MAE) when the error rate is 10%/5%/1%/0.1%.

3.6.6 Sensitivity Analysis

In this section, we analyze the effect of four parameters: (i) de-quantization latency, (ii) quantization latency, (iii) the fraction of the outliers, and (iv) the location of the conversion in the memory hierarchy, in a system with 4 cores.

(i) De-quantization and (ii) Quantization latency. The conversion latency depends on many design parameters (e.g. technology size, parallel vs serial `Converter`, etc.). Figure 3.16(a) shows that for a de-quantization latency as large as 20 cycles, our proposed method achieves 1.38× speedup on average. The low sensitivity to the quantization latency stems from the fact that a quantized L2 block fills multiple de-quantized L1 blocks and consequently subsequent L1 hits amortize the additional conversion penalty on a L1 miss. The

speedup is *insensitive* to the quantization latency (Figure 3.16(b)) because quantization occurs on writeback requests which are out of critical path.

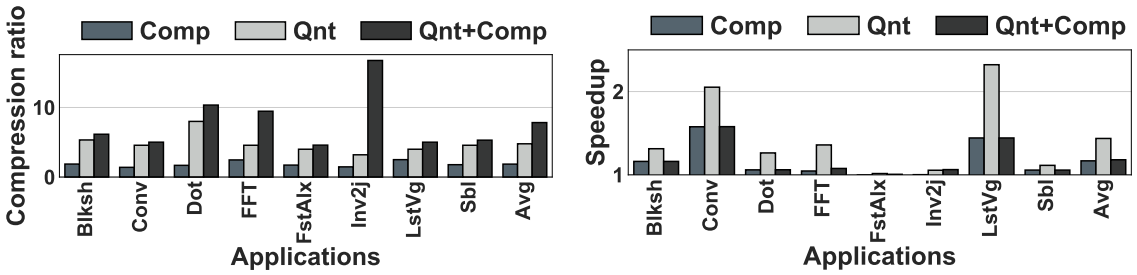
(iii) The fraction of the outliers. We find that the fraction of the outliers is very low (the highest rate belongs to Blackscholes and FFT where the fraction is 0.00001, 0.00002, respectively) that they impose a negligible overhead. In order to analyze the overhead of the outliers, we synthetically varied the fraction of outliers from 0.00001 to 0.1. Figure 3.16(c) shows that the fraction of outliers does not affect the performance except for a high fraction such as 0.1. The *insensitivity* to the fraction of outliers stems from the fact that the access to outliers is not on the critical path unless the outlier is in the missed L1 block (and not in the rest of unpacked blocks). Figure 3.16(d) shows that for some applications such as FFT (with relatively high L2 miss rate), the accesses to outliers may occur on the critical path and in that case, even a lower fraction such as 0.01 can significantly impact the performance.

(iv) Location of the conversion. So far, we evaluated the performance of our proposed method when data conversion occurs between L1 and L2. Figure 3.16(e) evaluates the alternative design, where conversion happens between L2 and L3. As a result, the conversion happens less frequently, decreasing the energy consumption. This figure shows that placing the conversion point between L2 and L3 provides on average, 1.22× and up to 1.4× speedup.

3.6.7 Quantization vs. Compression

The most related cache compression technique that takes into account the effect of data type and the effect of floating point is Hycomp [5].

Figure 3.17(a) illustrates that Hycomp yields only a compression ratio of 1.85 on average whereas our method itself yields the ratio of 4.78. Although, quantization on top of compression achieves a higher compression rate (7.83), a variable cache block size of compresssion will again require a complex mechanism for finding the location of cache blocks. We evaluated the effect of decoupling the tag array and data array in the last level cache. Figure3.17 (b) shows that the latency of access to the decoupled tag and data array nullifies the higher compression rate that can be achieved by employing compression on top of quantization. Figure 3.18 demonstrates that address translation by BitLocationFinder and corner case handling (Section 3.4.6) outperforms padding (which is proposed by LCP [60] to avoid the cost of decoupling tag array and data array), on average by 20% and up to 45%, in a 4-core system, where average bit-width is 18.



(a) Compression ratio (b) Speedup

Figure 3.17: Quantization vs. compression

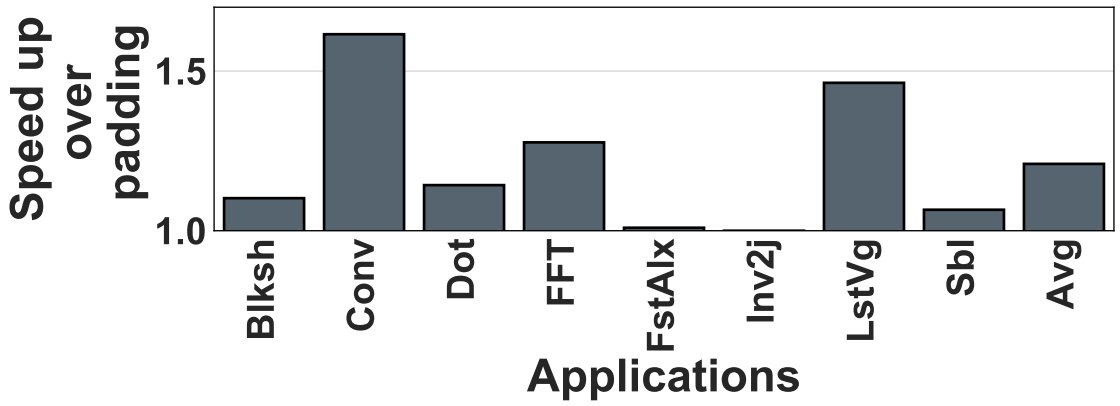


Figure 3.18: Padding (LCP) vs. BitLocationFinder

3.6.8 Software-based Quantization vs Hardware-based Quantization

Figure 3.19 demonstrates that our hardware-based design delivers 1.54× speedup compared to software-based approach, in a 4-core system. The speedup stems from less data movement and lower conversion overhead. We tried our best to manually optimize the code and perform conversion only when it is required. One of the benefits of our method is that the programmer does not need to think about optimizations, overflows, and outliers for every line of arithmetic computations in the code. In software, for applications such as Blackscholes we had to use standard 32-bit variables to cover the outliers that require 18 bits. With quantization in hardware we can employ 6-bit variables for Blacksholes and store outliers in their original format in a separate space.

3.6.9 Quantization vs. Prefetching

Unlike our method, prefetching can not alleviate the energy cost of data movement and do not increase the effective size of caches or decrease the memory bandwidth consumption. However, we can employ quantization in tandem with a prefetcher to hide the conversion latency. Prefetcher tracks physical addresses and does not need address translation. Therefore, we modified the prefetcher to record the *MetaDataID* for the previous

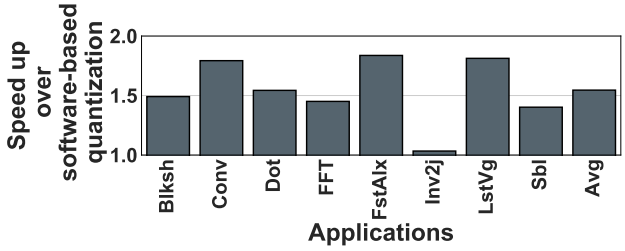


Figure 3.19: Quantization in hardware vs. software

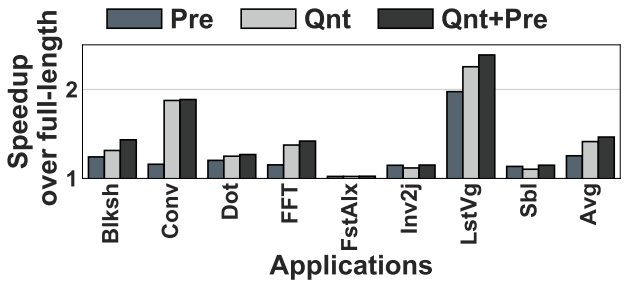


Figure 3.20: Quantization vs. prefetching

misses and use it for de-quantizing the prefetched blocks. Figure 3.20 shows that our method achieves on average 1.6× speedup over prefetcher. Figure 3.20 also demonstrates that our method along with prefetchers can achieve on average 1.46× speedup over the baseline system.

3.6.10 Hardware Overhead

We design and synthesize the modules using an industry-standard 1xFinFET technology with foundry models.The total area of our modules, per core, is 0.012455 mm^2. Accordingly, 16 modules in a 16-core processor impose 0.2% area overhead (we scaled the area of the processor in 22 nm technology, obtained from McPAT [82], to derive the area of the processor in 1xFinFET technology). The total overhead is 0.25% of the original processor.

3.7 Related Work

Bit-width Reduction Techniques. Several studies employed the OLSB method for ASIC and FPGA designs to customize the *bit-width* [8, 9]. Prior works also show the energy reduction of quantization with arbitrary *bit-width* in acceleretor design and provide a framework for deciding the *bit-width* [10, 11]. However, these solutions cannot be adopted in the general-purpose processors. The only general-purpose solution, ACME [1] employs the OLSB method which has lower accuracy and lower performance than quantization.

Software-based Quantization for Specific Applications.

Due to high overhead, software-based quantization is not a popular technique unless in three following cases: (i) to compress data before storing in storage [86], (ii) to compress data before offloading data to accelerators [4, 87], and (ii) in some cheap embedded processors [88, 89] that cannot accommodate floating point ALU, where the cost of conversions is less than the emulation of floating point operations on integer ALU and programmer's effort for manipulating integers pays off.

Approximate cache techniques. Cache approximation techniques [90, 91] also reduce the cost of movement and storage of data. We do not compare against any other approximate cache techniques, as ACME [1] (to which we have compared our method in Section 3.6.5) shows performance improvement over the prior approximate cache techniques, Doppelganger [90].

Chapter 4

Fulcrum: a Simplified Control and Access Mechanism toward Flexible and Practical In-situ Accelerators

4.1 Introduction

The energy consumption and execution time of applications with low computational intensity (low computation per datum) is mainly due to the high cost of data movement [92, 93]. This is illustrated by a prior work that shows the energy cost of fetching a 32-bit word of data from off-chip DRAM is 6400× higher than an ADD operation in 45 nm technology [4]. We refer to applications with low computational intensity as memory-intensive applications. In the Big Data era, many applications, such as data analytics, scientific computing, graph processing, and machine learning, are memory-intensive.

To minimize the cost of data movement, recent studies have explored the possibility of processing data near or even inside the memory (PIM), for example, TOP-PIM [94], DRISA [2], Ambit [12], SCOPE [95], PRIME [29], ISAAC [28], and TOM [96].

Memory is designed in a multi-level hierarchy, consisting of vaults (in 3D stacked memories), banks, subarrays, etc. (more details in Section 6.2). PIM designs vary based on the level of the memory hierarchy at which the computation is performed. In-situ [2, 12] computing is one of the most aggressive forms of PIM that processes data in the row buffer, very close to the subarrays in which the data are stored. The subarray level provides a high throughput for two reasons: (i) the low latency of each access at the subarray level, and

(ii) high subarray-level parallelism (e.g., Hybrid Memory Cube (HMC) has 512 banks [19], and assuming 32 subarrays per bank, the HMC will have 16384 subarrays.).

In-memory processing architectures are not limited to conventional DRAM configurations and interfaces (e.g., DIMM or HBM). Some prior in-situ approaches [95, 2, 29, 28, 97] target their design as a peripheral-attached accelerator instead of as a host memory to avoid the tight area and cost constraints for commodity DRAM. Indeed, these designs can be thought of as high-throughput accelerator architectures that happen to use DRAM as the most effective technology, based on parallelism, proximity to data, and overall capacity.

DRAM, compared to SRAM/eDRAM, provides a higher capacity for the accelerator. Compared to NVM, DRAM provides a lower latency. DRAM is also more tolerant to frequent writing of partial results. Therefore, in this paper, we focus on DRAM-based in-situ accelerators. Although computation in DRAM technology is inefficient [98] compared to a traditional logic process, the low computation requirement of the memory-intensive applications justifies the slower computations in DRAM-based in-situ accelerators.

In this project, we identified four problems that limit the benefit of current in-situ computing approaches.

The first limitation is the lack of flexibility for supporting a wide range of applications, which in turn, limits the market for the product. Prior in-situ approaches [2, 12] perform the same operation in all subarrays and on all bytes (e.g., 256 bytes) of the row buffer (row-wide operations). As a result, they cannot efficiently support operations with data dependencies along the row buffer or operations with a condition or predicate. For example, prior works cannot efficiently support reduction and scans (which are more efficient with serial operations along the row buffer), database operations such as Sort, FilterByKey, and FilterByPredicate (which require filtering based on predicate), or sparse operations (which require multiplications and accumulations only for nonzero values with matched indexes). Unfortunately, enabling operations with data dependency or operations based on a predicate, at a subarray level, using the traditional control and access mechanism is not practical (the area of a simple in-order core, such as ARM Cortex-A35 with 8KB of cache, excluding the SIMD units and scaled to 22 nm, is 25× larger than the area of a subarray).

The second issue is the capacity of the accelerator, which can be limited by the hardware overhead of the computing elements. At the subarray level, we read an entire row at once and store it in the row buffer. A processing unit with the capability of performing bitwise operations, addition, and multiplication (integer and single-precision floating point) on all bytes of the row buffer is at least 52× larger than the subarray. Thus, in-situ approaches only employ bitwise ALUs. It means any other operation should be emulated using multiple bitwise operations, requiring multiple subarray accesses and hundreds of cycles. We will show that the cost of emulating complex operations using bitwise ALU is higher than the cost of moving data out of the vaults. Some in-situ approaches use the analog computation capability of memory cells of

ReRAM-based non-volatile memories (NVM) for multiplication and addition, without requiring any adder or multiplier [97, 28, 29]. However, this type of in-situ computation requires analog-to-digital converters (ADC), which also incur a significant hardware overhead. For example, the area overhead of ADCs in [97] is 45× higher than the area of the subarray. More importantly, analog computing introduces a high error rate and therefore, these accelerators suit applications that are highly tolerant of error, such as deep learning applications [29, 28]

The third problem is the physical layout of words in DRAM. The subarray itself is typically divided into smaller units, called mats. Due to circuit design constraints (which are explained in Section 4.2.4), in current memory designs, every four bits of a 32-bit word is stored in a separate mat. We refer to this as mat interleaving. Accordingly, prior in-situ approaches [2, 12] perform computation only on 1-bit, 2-bit, and 4-bit values.

The fourth limitation is inefficient peripheral logic. Many applications require sharing values among subarrays. For example, in matrix-vector multiplication, we can map each row of the matrix to a subarray. To perform matrix-vector multiplication, all subarrays need the values of the vector. A variety of applications require data movement among subarrays. For example, for the Sort application, the data is partitioned among subarrays. Then, all sorted partitions (which reside in different subarrays) should be merged. The merge requires inter-subarray data movement. Finally, many applications require parallel and independent column selection for each subarray. For example, in FilterByPredicate, in every subarray, we check a condition on each column and store the data in the row buffer, only if the condition is met. Therefore, different subarrays may write into different columns of the row buffer. Current peripheral logic serializes movement of shared values, inter-subarray movement, and column selection in different subarrays (more details in Section 4.2.3).

To resolve the aforementioned problems, we propose *Fulcrum*, where we rethink the design of in-situ accelerators to increase flexibility, practicality, and efficiency.

First, we propose a new lightweight access and control unit that processes data sequentially and determines the next operation based on the outcome of the previous operation (if necessary).

Second, we accommodate a processing unit capable of 32-bit addition, subtraction, and multiplication (in addition to bitwise operations) in the subarray level and limit the hardware overhead by performing operations on only a word of the row buffer at a time. Although this processing unit processes only a fraction (1/64) of the row buffer per cycle, it can provide a significant performance improvement overall. Our evaluation shows that performing complex operations on a subset of the row buffer outperforms prior in-situ approaches that perform computation on the whole row buffer but emulate complex operations by multiple bitwise operations.

Third, we slightly modified the mat-interleaving circuits to transfer all bits of a word to the side of a subarray (reuniting interleaved bits) so that all bits are physically close to each other. Since we only process one word at each cycle, the circuits for reuniting one word do not impose significant hardware overhead.

Fourth, we optimized peripheral logic for computation. To satisfy the data-sharing requirement, we enable broadcasting. For applications with inter-subarray data movement requirement, we employ a prior work [3] (Low-cost Inter-linked SubArrays (LISA)) that transfers a whole row from one subarray to another at once, reducing the overhead of inter-subarray data movement. For applications with independent column access requirements, we enable a light-weight independent column selection mechanism through storing one-hot-encoded values of the column address in latches.

Our processing unit in each subarray has four parts: (i) Walkers, that provide sequential access with our lightweight column-selection mechanism. Walkers store the input operands of the computation (which are read from the memory array) or the output of the computation (to be written in the memory subarray), (ii) a small programmable instruction buffer, where we store the pre-decoded signals for the computation, (iii) a simple controller that determines the next operation and direction of the sequential access, and (iv) a single-word ALU. We show the flexibility of our design by mapping important kernels from different domains such as linear algebra, machine learning, deep learning, database management system (DBMS), as well as sparse matrix-vector multiplication and sparse matrix-matrix multiplication that appears in scientific computing, graph processing, and deep learning.

We integrate Fulcrum with CXL [99], a new class of interconnect that extends PCIe to significantly reduce the latency of access to the data in peripheral devices. Therefore, CXL provides high bandwidth and high power delivery of PCIe while supporting memory-like access to the data stored in Fulcrum. The low access latency enables communication through a single memory model. As a result, CXL can connect multiple in-memory accelerators, such as Fulcrum, and create a pool of disaggregated in-memory accelerators, providing scalability. The memory-like access of CXL also benefits workloads with phase behavior, where some phase is suitable for in-situ computing while other phases are not. In these cases, the host or other devices can access the output of Fulcrum with the latency in the order of latency of accessing normal memory.

Our paper makes the following contributions:

- Broadening the range of supported applications by in-situ accelerators
- Optimizing peripheral logic
- Optimizing mapping of important memory-intensive kernels and applications to Fulcrum.

- Releasing the source code of three artifacts: (i) MoveProf, a tool that uses profiled performance metrics for evaluating the cost of data movement, (ii) InSituBench, an in-situ computing benchmark suite, and (iii) the RTL code of our proposed method.
- Integrating Fulcrum with CXL to increase the power budget and enable scalability and disaggregation.

We evaluated our method against three approaches: (i) a prior work on DRAM-based in-situ computing (DRISA [2]), (ii) a server-class GPU that has three stacks of high-bandwidth memory (NVIDIA P100 with HBM2), and (iii) an ideal model of a GPU, where we take into account only the cost of data movement between GPU and the memory, assuming zero overhead for on-chip data movement and computation. Our evaluation shows that, for memory-intensive applications, our method delivers, on average (up to), 70 (228)× speedup per memory stack over the GPU. Our area evaluation shows that an 8GB integer Fulcrum (which supports bitwise operations and integer addition and multiplication) requires $51.74mm^2$ (8 layers) and single-precision float Fulcrum (integer and bitwise functionality, plus floating-point addition and multiplication) require $55.26mm^2$. The GPU die size of the NVIDIA P100 is $601mm^2$. Accordingly, Fulcrum provides up to 839× higher throughput per area than the NVIDIA P100.

4.2 Problem statement

In order to reduce the cost of data movement for tasks that are data-intensive and have locality within a row or subarray, prior works have proposed in-situ computing, where we perform computation on the row buffer, and therefore, there is no need to move data out of the subarray. We identified four problems that have hindered adoption and realization of in-situ approaches.

4.2.1 Lack of flexibility

Recent in-situ approaches employed non-flexible row-wide operations. As a result, they cannot support operations with any form of dependency along the row buffer. For example, in operations such as Scan, the value of each partial sum depends on the value of the previous partial sum. They also cannot support algorithms that check a condition on a value and perform a different operation based on the outcome of the condition. For example, radix sort is an algorithm that sorts data by grouping values by the individual digits, which share the same significant position. Each iteration of this sort algorithm packs values into two different buffers. The target buffer for the value is determined by the digit that is being processed at that iteration. Another example is sparse matrix-vector multiplication, where we often store the non-zero values

next to their indices (instead of wasting the capacity by storing the whole matrix with mostly zero values). Consequently, we only perform multiplication and accumulation on non-zero values, whose index matches.

4.2.2 Lack of support for complex operations

Prior works have evaluated a spectrum of in-situ computing. Seshadri et al. [12], evaluated row-wide bitwise operations using computation capability of bitlines without adding any extra gate, realized by activating two rows at the cost of destroying the values in both rows, requiring extra copies before each operation. Li et al. [2], evaluated row-wide bitwise ALUs, shifters, and latches (the latches eliminate the extra copies), emulating 4-bit addition and 2-bit multiplication using bitwise ALUs. They also evaluated adding row-wide 4-bit adders to the row buffer and reported that this increases the area by 100%. Unfortunately, emulating complex operations such as addition or multiplication using bitwise ALUs requires reading and writing multiple rows. Since row activation is very costly, the energy consumption of row activation for emulating complex operations by bitwise operations surpasses the energy consumption of sending data to the logic layer, as shown in Figure 4.1.

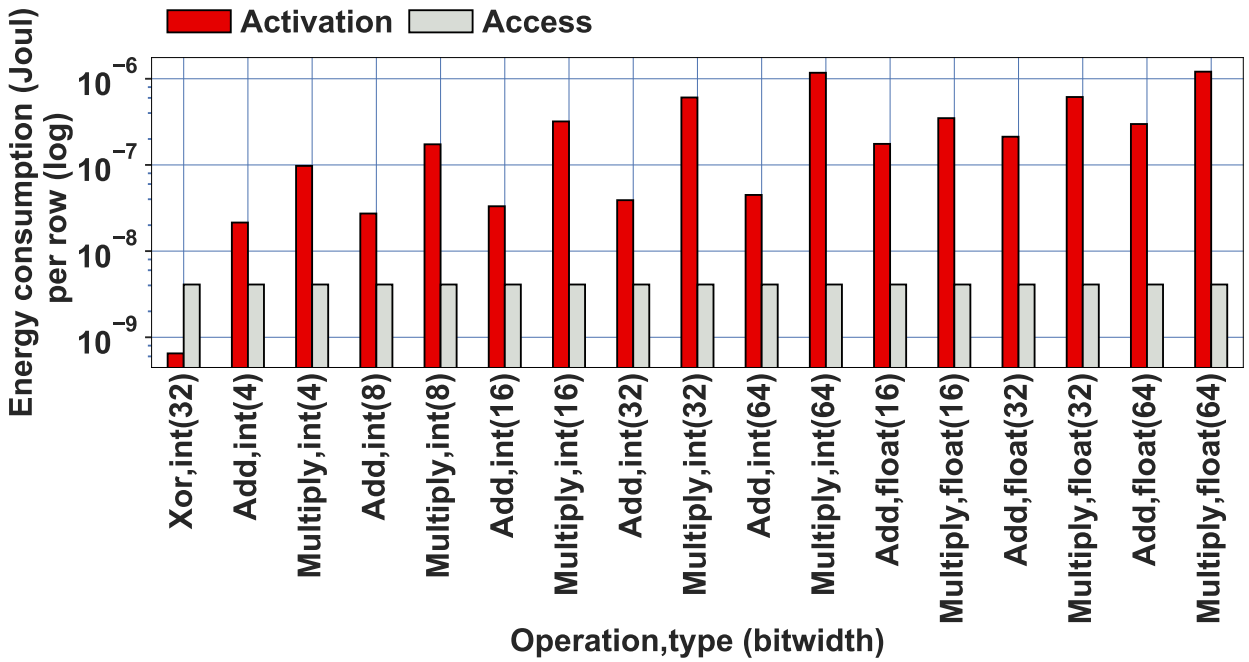


Figure 4.1: Energy consumption of accessing a row in the logic layer vs. energy consumption of multiple row activations, required for emulating complex operations by bitwise operations

4.2.3 Inefficient peripheral logic

Currently, there are three shared resources in the design of the memory: (i) the TSVs, shared among all layers in a vault (as explained in Section 6.2 and shown in Figure 2.1), (ii) the shared CSLs, and (iii) the

GDLs, shared among all subarrays in a bank. With shared CSL, we can only select one column from a whole bank at a time and with shared GDLs we can transfer only one column at a time. As a result, the current peripheral logic and interconnect limit the performance of in-situ approaches in three ways: (i) although they act as a shared bus, they are not capable of broadcasting values for efficient data sharing, (ii) narrow GDLs are the only means for movement of an entire row from one subarray to another, (ii) the peripheral logic for column access (the column decoder) is shared among all subarrays (Figure 2.1 (b)). This shared peripheral logic limits the flexibility and parallelism of any potential in-situ approach that requires independent and parallel column access to the row buffer of individual subarrays.

4.2.4 Interleaving

In current commercial DRAMs, we have two types of interleaving: (i) mat interleaving, and (ii) subarray interleaving. Mat interleaving is shown in figure 4.2, where each subarray is divided into multiple mats [100, 101]. The GDLs are distributed among mats, and each mat has 4 bits of the GDLs. Therefore, for selecting the same column from all mats, CSLs are repeated for each mat. Pass transistors (PTs) receive CSLs, select a column, and place it on LDLs. This design is called mat interleaving and is efficient for random column access, as it reduces the LDLs's latency (LDLs in Figure 4.2 are shorter than LDLs in Figure 2.1 (b)). Without mat interleaving, LDLs become wide and long, where the latency of the last column is much longer than the latency of the first column.

The second type of interleaving is subarray interleaving or open-bitline architecture [3, 102]. Since the size of a sense amplifier is larger than a cell [102], modern DRAM designs accommodate only as many as sense amplifiers in a row to sense half a row of cells. To sense the entire row of cells, each subarray has bitlines that connect two rows of sense amplifiers, one above and one below the subarray.

As a side benefit, mat interleaving and subarray interleaving make the memory more robust against multiple-bit upset, where soft errors change the value of adjacent cells. In fact, when bits in a column are not physically close to each other, multiple-bit upset only changes one bit from a column and then error detection mechanisms (which can detect one error) can detect the error. Therefore, keeping the current interleaving and not changing the layout is desirable.

However, with interleaving, row-wide computation on more than 4-bit values is impractical, as the result of an addition and multiplication in each 4 bits of the output depends on the values in other mats. With row-wide operations, the circuits for reuniting the interleaved bits impose a significant hardware overhead as many wires cross each other.

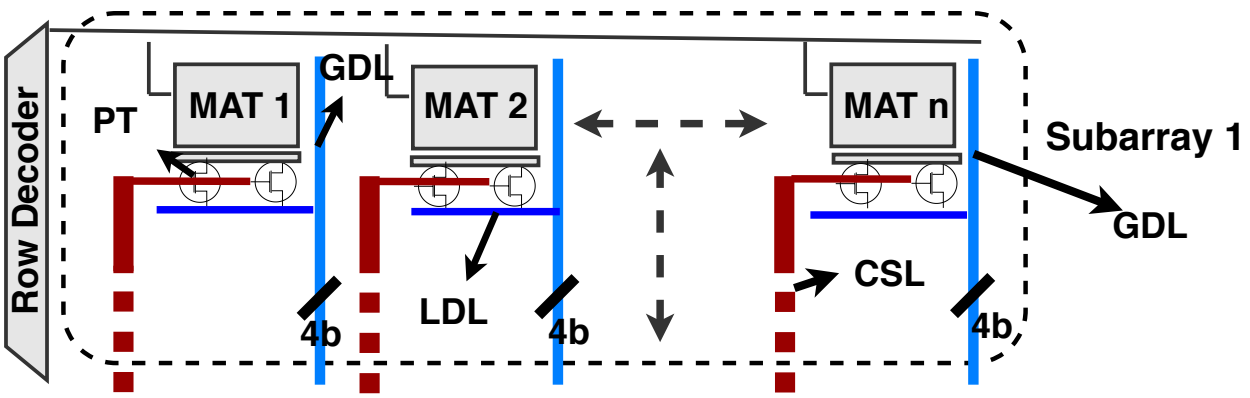


Figure 4.2: Interleaving a word among mats

4.3 Key ideas

The key insight of this paper is that a narrow ALU with adder and multiplier, placed at the edge of each pair of subarrays, can outperform row-wide bitwise ALUs, because bitwise ALUs suffer from the drawbacks previously described. We show that these capabilities are important for a variety of computational kernels. Figure 4.3 (a) shows the architecture of our in-situ processing unit, which has two parts: (i) Walkers and (ii) AddressLess Processing Unit (ALPU). The Walkers are implemented by three rows of latches that are connected through a bus similar to LDLs (implementation details in 4.4).

The ALPU itself comprises four components: (i) a controller, (ii) three temp registers, (iii) an ALU, and (iv) an instruction buffer.

In this section, we explain the role of these components in resolving prior in-situ approaches' limitations and elaborate on implementation detail in Section 4.4.

4.3.1 A simplified control and access mechanism

Since a narrow ALU only processes a word of row buffer, we need a sequential access mechanism for selecting consecutive words. Due to the significant hardware overhead, employing a core with traditional access and control mechanism, for sequentially selecting a word from a row buffer at the subarray level, is impractical. We could give up subarray-level parallelism and employ only one traditional core per bank to limit the hardware overhead. Unfortunately, other than losing subarray-level parallelism, this solution imposes a significant overhead for control and access. Since logic in DRAM layers is slow, and the cost of data movement is low, unlike far-memory processors, this overhead comprises a significant portion of total energy consumption and execution time. Figure 4.4 (top) illustrates the control and access overhead of a traditional core at the bank-level for adding two vectors and storing the result in a third vector. In this example, the core

generates an address for each element of the three vectors, imposing access overhead. Since the core is placed at the edge of each bank, the decoded addresses should be sent to the subarray through CSLs (Section 6.2 and Figure 2.1). Data read from the subarray also should move toward the core through GDLs, imposing data movement overhead. The decoding of all these instructions, branch prediction, and checking for data dependency in the pipeline of traditional cores impose control overheads. Despite the significant overhead, such a control and access mechanism provides full flexibility.

Our proposed method provides a tradeoff between flexibility and the overhead of control and access. In fact, while we enable operations with data dependency and operation based on predicates, we avoid the overhead of sophisticated control mechanisms and the overhead of accessing data by address. We observed that for almost any memory technology, an entire row is read/activated at once and stored in a buffer. We introduced Walkers, where each Walker either captures a row of input operands (read from the subarray) or stores a row of target variables (before being written to the subarray). We read/write to/from these rows sequentially and implement the sequential accesses using shifting of a one-hot-encoded value that determines which column of the row should be selected to be placed on the bus. Our simple controller determines the direction of shifts in each Walker and also determines the next operation based on the outcome of the previous operation, providing flexibility. Accordingly, for example, our simplified access and control mechanism performs an addition of two vectors by iteration of an instruction over the row buffer, similar to the instruction shown in Figure 4.4 (down) (Section 4.4 presents the exact format of Fulcrum's instructions and discuses how our hardware modules control loading new rows of each vector to Walkers).

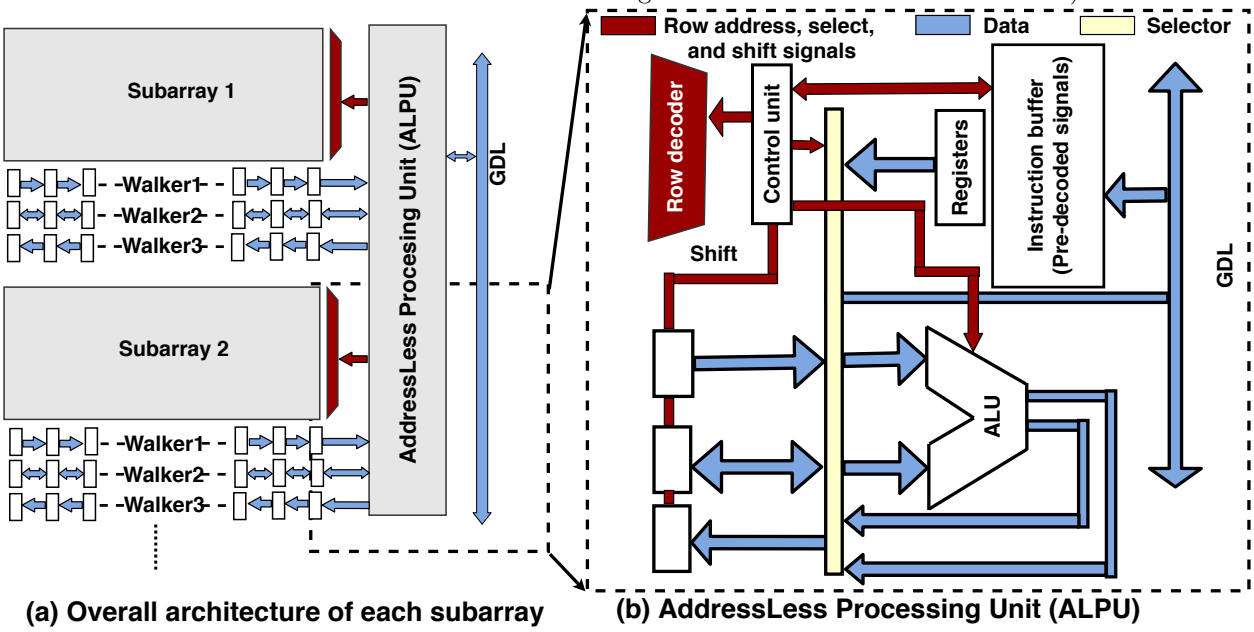


Figure 4.3: Overall architecture: (a) three Walkers per subarray and one ALPU per every two subarrays, (b) ALPU comprises a controller, an ALU, an instruction buffer, and few temp registers

(a) Traditional control and access mechanisms

```
1. RA=RA+1 // address generation for vector a[i] + control
2. RB=RB+1 // address generation for vector b[i] + control
3. RC=RC+1 // address generation for vector c[i] + control
4. Read RA, X1 // decoded address movement on CSLs
                  //  data movement on GDL + control
5. Read RB, X2 // data movement on GDL + control
6. X3=X2+X1   // the actual computation + control
7. Store X3, RC  //decoded address movement on CSLs
8. i=i+1          // iteration counting + control
9. If (i< 1000000) Jump 1 //  iteration checking+ control
```

(b) Fulcrum

```
Shift_Read a[i], shift_Read b[i], Shift_Write c[i] , c[i]=a[i]+b[i]
```

Figure 4.4: A traditional control and access mechanism vs. Fulcrum

4.3.2 Narrow and simple ALU

Our sequential access to Walkers enables processing only one word (one column) at a time, and consequently we do not need row-wide ALUs. We observed that addition, comparison, multiplication, and bitwise operations are the most common operations that appear in modern memory-intensive applications. Therefore, we included a single-word ALU, which supports these common operations. The input operands of the ALU can come from one of the four resources: (i) the value sequentially accessed from one of the Walkers, (ii) temp registers, (iii) the GDLs, or (iv) one of the outputs of the ALU (our controller supports two operations in one cycle). Section 6.4 explains that although our ALU is narrow, it outperforms row-wide bitwise operations and supports modern memory-intensive applications.

4.3.3 Efficient peripheral logic

We have introduced minor modifications in the peripheral logic to increase flexibility, parallelism, and efficiency of data movement for our method. First, we added a broadcasting command, by which every processing unit receives and captures the data on shared buses. Second, we build upon Low-cost Inter-linked SubArrays (LISA) [3] to transfer an entire row at once to any other subarray in the same bank (otherwise we had to transfer the entire row, column by column, through the narrow GDL bus). Third, for independent column access in each subarray, instead of using column decoder and column address buses, which are shared among

subarrays, we employ column selection latches in each subarray, where we store a one-hot-encoded value that determines the selected column. In each cycle, based on the outcome of the previous operation, the controller decides in which direction the on-hot-encoded value should be shifted.

4.3.4 Reuniting interleaved bits

Unlike reuniting the whole row, reuniting one word is possible through a slight modification of the current mat interleaving circuits. Therefore, we can transfer and reunite interleaved bits of a word at the side of the subarray to perform arithmetic operations.

To resolve subarray interleaving, we simply use only one processing unit per two subarrays. Figure 4.3 shows that per every two subarrays, we only have one ALPU.

To resolve the mat interleaving, we propose two solutions. Our first solution is to change the layout and completely remove the mat interleaving (if the target application does not need efficient random column access or is resilient against soft errors). As a side benefit of eliminating mat-interleaving, we will save the area overhead of CLSs and columns selection logics (repeated for each mat). The second solution is to keep

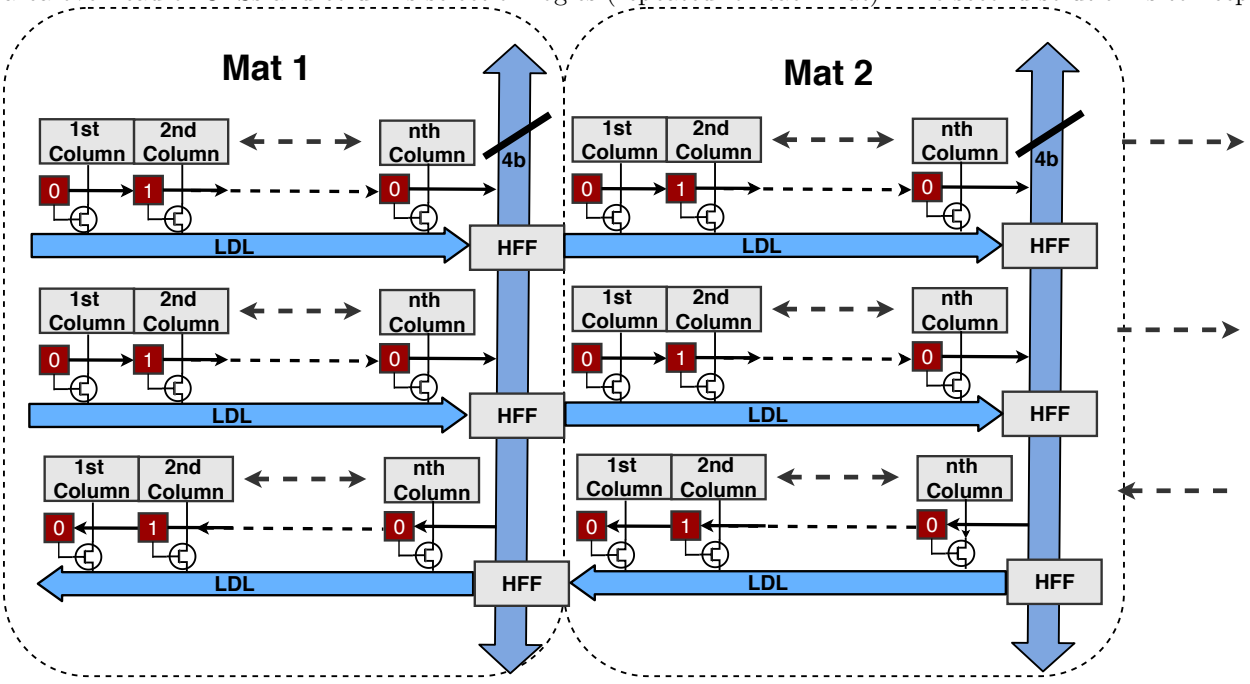


Figure 4.5: Reuniting interleaved bits

the mat interleaving. In the traditional design of memory, each segment of the LDL corresponding to each mat is connected to four flip-flops (called helper flip flops (HFF)). We connect these flip flops to the segment of the LDL of the adjacent mat and form a pipeline so that we can transfer all values to the side of the subarray in a pipeline fashion. In a memory structure with four mats per subarray, this pipeline requires four cycles to transfer 32 bits (16 bits from the upper Walker and 16 bits from the lower Walker). If we

use the TSV for sending the clock signal, the clock cycle should be at least twice as long as the latency of the TSV (according to CACTI-3DD, the latency of TSV with eight memory layers and one logic layer is more than 4.4 ns). This will significantly degrades our throughput. To resolve this issue, we employed segmented TSV [20, 21] (explained in Section 6.2) for the clock signal (according to CACTI-3DD, the latency of segmented TSV is 0.3 ns).

4.4 Implementation detail

This section elaborates on hardware and software implementation.

4.4.1 Hardware

This section provides more implementation details about six hardware components: (i) Walkers, (ii) the controller, (iii) the instruction buffer, (iv) our in-logic layer components, (v) interconnect, and (vi) the walker renamer.

The Walkers

Walkers provide sequential accesses. We have two options for implementing the Walkers. The first option is changing the layout, employing shift registers (or shift latches, implemented by ping-pong shifting [23]), and accessing the row sequentially by shifting the values. The second solution is keeping the interleaved layout, employing the structure of traditional row buffers and local buses (similar to LDLs, explained in Section 6.2) along with a column-selection mechanism that selects a column to be placed on each Walker's LDL. In addition to keeping the interleaved format, the second solution has two side advantages: (i) enabling sequential read and write in both directions (with the first option, we can only read by shifting to the right and write by shifting to the left (in Figure 4.3)), and (ii) consuming less energy (with first option, per each shift, there will be value transition in all latches, whereas with the second solution, only the value of the bus and only one latch changes).

The only difference between the structure of the traditional row buffer and our Walkers is the column-selection mechanism. As we explained in Section 6.2 and Section 4.2, traditional memories share the peripheral logic for column selections. Figure 2.1 (b) shows that the CLSs (on which the decoded column address is placed), are shared among subarrays. Figure 4.2 shows that CLSs are repeated for each mat. To support operations based on predicates, we need independent column access for all Walkers and subarrays. Hence, we introduced column-selection latches where we store the one-hot-encoded value of a column and shift the value to access the next column, without requiring one column decoder per Walker, per subarray, and per mat.

The controller

Our controller employs a few counters: (i) a 6-bit counter (6 = log2 of the number of words in a row (64)) per Walker for detecting a fully-accessed Walker (fully read or written). Each Walker has a 2-bit latch that determines to which Walker we should switch and rename when the Walker is fully accessed (elaborated in

Section 4.4.1), (ii) a 4-bit counter (4 = ceil of log2 of the number of wait cycles (9)) for counting the wait time for a new row to be read from the subarray and be stored in a Walker, or for a Walker to be written to the subarray, and (iii) three 11-bit (11 = log 2 of the number of rows in each pair of subarrays (2048)) row counters which are initialized to the row address of the beginning of the data and will be compared against the end of the data in the subarray.

The instruction buffer

Figure 4.6 illustrates the format of each entry of the instruction buffer. This format allows two operations at the same time and has the following fields: (i) NextPc1 and NextPc2 that determine the program counter of the next instruction, (ii) NextPc_Cond determines the condition under which the controller switches to instruction determined by NextPc1 (otherwise, it switches to NextPc2). When NextPc1 equals NextPc2, the NextPc_Cond is used for determining which comparison flag should be the input bit of the bitwise shift operation, (iii) opCode1 and opCode2 are the operation codes of each operation, (iv) Src1Op1, Src2Op1, Src1Op2, and Src2Op2 select a source for each input of the operation, (v) ShiftCon1, ShiftCon2, and ShiftCon3 specify the condition under which the corresponding Walker should be shifted, (vi) ShiftDir1, ShiftDir2, and ShiftDir3 determine the direction of shifts in each row, (vii) repeat filed is the number of repeat before shifting when any of the shift conditions are IF_REPEAT_ENDS_SHIFT", and (vii) OutSrc selects the value shifted to the destination row among the two operation outputs and the two outputs of the first two Walkers. Although for the evaluated benchmarks an instruction buffer with four entries is enough, we also evaluated our area overhead with an instruction buffer that has eight entries.

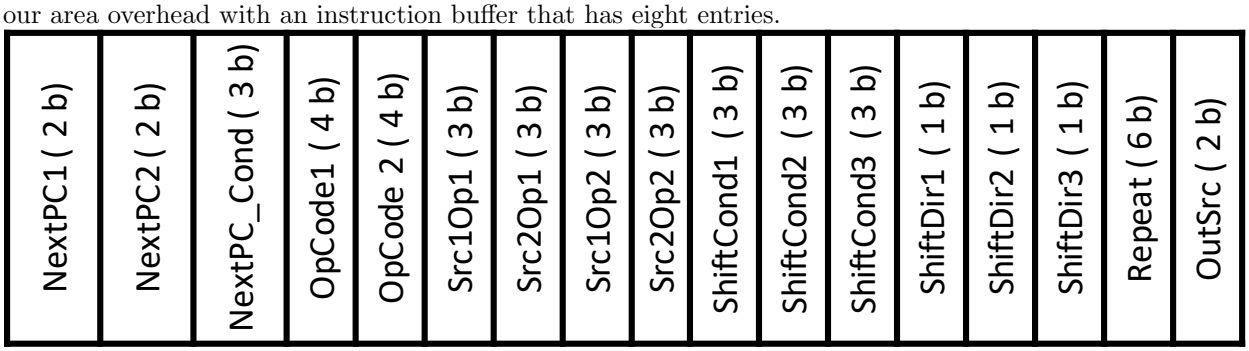


Figure 4.6: The format of instructions

In-Logic layer components

For in-logic layer operations, we use an ARM Cortex-A35 which is used in prior works [84] as the processing unit in logic layer (because it has low power consumption), along with a 128 KB buffer for buffering shared values.

Interconnect

Fulcrum can be integrated into a system in two ways. The first option is to integrate Fulcrum with CPU/GPU through current 3D interfaces. For this option, we have to lower the power budget to 10 Watt [94, 84] (we evaluate Fulcrum under different power budgets in Section 6.4). The second option is to use CXL [99] interface (similar to GPU and FPGA), which allows a higher power budget. This option allows a memory-like access to the data stored in Fulcrum. As a result, for workloads that have phase behavior, where some phases are suitable for in-situ while other phases are not, the host, GPU, FPGA, or any other device can access the output of Fulcrum (output of in-situ phases).

The Walker renamer

Fulcrum exploits broadcasting for reducing the cost of data movement. To overlap computation and broadcasting, all subarrays should work in lockstep so that the broadcasted value is used for computation and can be discarded in the next cycle, eliminating the need for storing broadcasted values. However, in some applications, the processing time for each Walker in each subarray might vary, hindering the lockstep computation. We propose to exploit Walker renaming to solve this problem. As an example, we explain the role of Walker renaming in SPMV.

To represent sparse matrices (where most of the values are zero), we can employ a few formats. One of the most popular formats is the compressed sparse row (CSR) format, which represents a matrix M by three arrays containing: (i) nonzero values, (ii) the positions of the start of the rows, and (iii) column indices. A naive implementation lays out the three vectors in three different rows and uses all Walkers. Since the values of the vector are being broadcasted, when a controller detects a fully accessed Walker in any of the subarrays, the process, in all subarrays, should wait until a new row is read into the Walker. To avoid this overhead, we place each non-zero value and its corresponding column index subsequently in the same array. This way, we only need one Walker for computation. Therefore, for example, while Walker A is being processed, another row can be captured in Walker B. When Walker A is fully accessed, the computation can continue by processing Walker B. However, the ALPU's instruction buffer is programmed to process Walker A. Here Walker renaming can help. As explained in Section 4.4.1, when Walker A is fully accessed, Walker B will be renamed to Walker A so that computation can continue with the same ALPU instruction.

4.4.2 Software

In this section, we discuss programming, data placement, and high-level programming.

Programming

Fulcrum's programs comprise two parts: (i) the in-logic layer portion, and (ii) the ALPUs' portion.

Our in-logic layer programs interact with vault controllers for generating commands for setting ALPU's registers and instruction buffer, sending broadcast values, and collecting the partial results. It also reduces the partial results or performs specific functions on intermediate results. Our in-logic layer core is an ARM Cortex-A35 and can be programmed by high-level programming languages such as C++.

To program the ALPU, we used the low-level instructions explained in Section 4.4.1 (Figure 4.6). Our online repository [103] contains ALPU programs for the evaluated applications. A non-expert programmer can easily use ALPU libraries, written by experts (similar to machine-learning users with no CUDA knowledge that are using NVIDIA libraries).

Therefore, a Fulcrum kernel call first loads the in-logic layer program. The in-logic layer program generates commands for the vault controller to load the ALPU programs and other ALPU settings and start the computation.

Data placement

The layout of data highly affects the performance benefit of Fulcrum. The data should be partitioned and laid out carefully to enable exploiting subarray-level parallelism, broadcasting, and the light-weight sequential access mechanism. For example, in matrix-vector multiplication, we use a row-oriented layout for the matrix and map each row of a matrix to one pair of subarrays. In each cycle, we broadcast one element of the vector to all ALPUs, and each ALPU multiplies the broadcasted value by the corresponding element of the row of the matrix. To choose the best strategy for placing data in the desirable layout, we categorize data as either: (i) long-term and (ii) temporary resident data. The first category resides in Fulcrum for a long time, but the second group is the input of the application or the intermediate results that reside in Fulcrum temporarily. For example, DNN algorithms are composed of several layers. The core of computation in each layer is a matrix-vector multiplication, where a matrix of weights are multiplied by a vector of activations (for batch size of one). The output of each layer is a vector, which is the activation vector for the next layer. The matrix of weights can reside in Fulcrum for a long time. However, the activation vectors, which are the output of each layer, only reside in Fulcrum for a short time. Machine learning models such as reference points in KNN or database tables are other examples of long-term resident data. The query points in KNN are examples of temporary resident data.

For long-term data, we assume an offload paradigm for both the 3D and CXL deployments. So an API similar to CUDA's API (cudaMemcpy()) manages the data transfer. Mapping the address space to DRAM

rows, banks, and subarrays is configurable in the memory controller [19]. Therefore, by copying data in a specific address, the programmer can place data in the desirable layout. For this group, we can ignore the overhead of laying out the data as this is a one-time cost.

The temporary resident data itself can be categorized into two groups. The first group are the data that we broadcast and do not need to be stored, such as the activation vectors or the query points in KNN. We store these values in our in-logic layer buffer. The space in this broadcasting buffer has also a memory address. If these data are generated outside Fulcrum (for example, in GPU, in the previous phase of the applications which are not suitable for in-situ computing), an API copies these data to the in-logic layer buffer. If these data are generated inside Fulcrum, our in-logic layer core's program collects these data from the memory and stores it in the in-logic layer buffer. The second group are the data that cannot be broadcast. If these data are generated outside Fulcrum, we propose to employ on-the-fly layout optimization methods [104] that are proposed for GPUs. If these data are generated inside Fulcrum but need to be laid out differently for the next phase, our in-logic layer core's program collects and changes the layout for the next phase.

High-level programming

We realize that a non-expert programmer will not write an assembly program for ALPUs. Our future work will develop a high-level programming language and a software stack. We hypothesize that a programming model similar to TensorFlow suits Fulcrum. Accordingly, we envision a software stack composed of two steps. The first step is to implement the important kernels of most commonly used libraries, such as cuBLAS [105], cuSPARS [106], and Thrust [107] and any other useful primitive such as Reduction, Scan, Sort, and Filter that are amenable for in-situ computing. The second step is to develop a programming model similar to TensorFlow([97]). The TensorFlow programs are Data-Flow Graphs (DFG) where each operator node can have multi-dimensional vectors, or tensors, as operands. The compiler transforms the input DFG into a collection of primitives and kernels which are implemented in step1. A similar approach is used for TPU and prior in-situ accelerators [97, 95].

4.5 Evaluation

In this section, we first describe our evaluation methodology. Second, we compare the performance of our method against three approaches: (i) a server-class GPU, (ii) a prior work on in-situ computing, and (iii) an ideal model of the GPU, where we only incorporate the cost of data movement. Third, we discuss applications' characteristics that affect the Fulcrum's performance and energy benefit. Fourth, we evaluate the effect

of each problem. Finally, we present area, energy, and power evaluation results, as well as performance evaluation under specific power budgets.

4.5.1 Methodology

We evaluated performance, area, and energy consumption of Fulcrum. For performance evaluation of Fulcrum, we divided applications into multiple phases, where each phase could either be an ALPU processing phase or an in-logic layer processing phase. We evaluated ALPU processing by modeling the ALPU's computation time, including both the row activation time and processing time. We also modeled the in-logic layer processing time, including data movement and partial-result calculations.

We evaluated Fulcrum with both integer and floating-point configurations: Integer Fulcrum is capable of integer addition and multiplication, as well as bitwise operations, whereas float Fulcrum adds single-precision floating-point addition and multiplication.

The major benefit of Fulcrum is reducing the cost of data movement. Accordingly, to abstract away from architectural details of GPUs, we also evaluated against an ideal model of the GPU, where we only incorporate the cost of data movement to and from the GPU's global memory. To this end, we measured the data that are read or written to the GPU's DRAM (using NVProf [108]) and divided the measured data movement by the raw bandwidth of the memory stack to obtain the performance cost of the data movement.

Our RTL and CACTI-3DD [101] evaluations show that Fulcrum can work at a frequency of 199 MHZ, in 22nm technology. We added slack of 21.5% (to incorporate the delay penalty of logic in DRAM technology [98]) and evaluated Fulcrum with 164 MHZ.

Table 5.1 lists the configuration of our evaluated systems, and Table 5.2 introduces our in-situ benchmark suite, InSituBench [109] (a combination of memory-intensive kernels, suitable for in-situ computing, from different domains). We selected Sort, Scan, Reduction, GEMM (matrix-matrix multiplication), and GEMV (matrix-vector multiplication) from the NVIDIA SDK benchmark [110]. We also included sparse matrix-vector (SMPV) and sparse matrix-matrix (SPMM), LSTM (a deep learning application), K-nearest neighbor (KNN) [111, 112] (a classical machine learning application), Scale, and AXPY (representatives of simple kernels). We also added FilterByKey, FilterByPredicate, Bitmap (from DBMS domain [113, 114, 115, 84]), and Xor (representative of bitwise kernels, used in bitmap indexing [116] and bitmap-based graph processing [117, 2, 25, 12]).

For area evaluation, we designed the ALPU in RTL and synthesized the modules using an industry-standard 1xFinFET technology with foundry models (in modern technologies the node number does not refer to any feature in the process, and foundries use slightly different conventions. We use 1x to denote the

Table 4.1: The evaluated applications

Application	Implementation	Operation	DRISA ?	input options
AXPY	cuBLAS [105]	add and multiply	Yes	–Num=1000000000
Bitmap	Thrust [107]	compare and bitwise shift	No	–Num=1000000000
FilterByKey	Thrust [107]	compare	No	–Num=100000000
FilterByPredicate	Thrust [107]	compare	No	–Num=1000000000
GEMM	cuBLAS [105]	add and multiply	Yes	–numRowA=25600 numColA=19200 –numRowB=19200 –numColB=12800
GEMV	cuBLAS [105]	add and multiply	Yes	–numRow=25600 –numCol=19200 –IsVector=1
KNN	Fast KNN [111, 112]	add and multiply	NO	–global_mem –ref_nb=100000 –query_nb=1
LSTM	cuDNN [118]	add and multiply	Yes	–seqLength=100 –numLayers=4 –hiddenSize=4096 –miniBatch=1
Reduction	NVIDIA SDK [110]	add	Yes	–Num=16777216
SPMM	cuSPARSE [106]	compare, add and multiply	No	–NumRowA=8192 –NumColA=100000 –NumColB=8192 –percentage=0.2
SPMV	cuSPARSE [106]	compare, add and multiply	No	–NumRow=8192 –NumCol=100000 –percentage=0.2
Scale	Thrust [107]	multiply	Yes	–Num=1000000000
Scan	NVIDIA SDK [110]	add	No	–Num=1073741824
Sort	NVIDIA SDK [110]	compare	No	–Num=10000000
Xor	Thrust [107]	bitwise	Yes	–Num=1000000000

14/16nm FinFET nodes offered by the foundry.). Then we scaled the area estimation (both pessimistic and optimistic) to 22 nm technology. We modeled the area of Walkers in CACTI-3DD [101].

To evaluate the energy consumption, we extracted the energy consumption of ALPU by RTL simulation, and used the energy consumption modeling of CACTI3DD [101] for Walkers. To evaluate the breakdown of energy consumption for the GPU, we developed MoveProf [119] that integrates NVIDIA's NVProf [108] with GPUWattch [120]. GPUWattch [120] uses RTL models for processing elements and CACTI [101] for memory elements. Therefore our evaluation of Fulcrum is comparable to GPUWattch [120].

Table 4.2: Configuration details for evaluated architectures

Component	Parameters
GPU	Tesla P100 [121], 12 GB memory 3 HBM2 memory stacks at 549 GB/s (183 GB/s per stack)
Fulcrum	technology:22 nm, 32 vaults 32 subarray, open-bitline structure 4 mats per subarray, 256 bytes per row 64 banks per layer, 8 memory layers, row cycle:50 ns, frequency:164 MHz In-logic layer: 128 KB SRAM-based FIFO ARM Cortex-A35
Ideal machine	HBM2, bandwidth:183 GB/s

4.5.2 Performance improvement over GPU

Figure 4.7 illustrates the throughput of Fulcrum over a server-class GPU, NVIDIA P100. This figure shows that Fulcrum outperforms the GPU, on average by 23.4×, and up to 76× (achieved for Bitmap). For applications such as Sort, with lower memory-intensity, the speedup is around one order of magnitude (8.8×). Applications such as GEMM, which can employ blocking to significantly increase locality, gain lower speedup (1.5×). NVIDIA P100 has three memory stacks and Fulcrum has one memory stack. Therefore, Fulcrum delivers, on average (up to), 70 (228)× speedup per memory stack over the GPU.

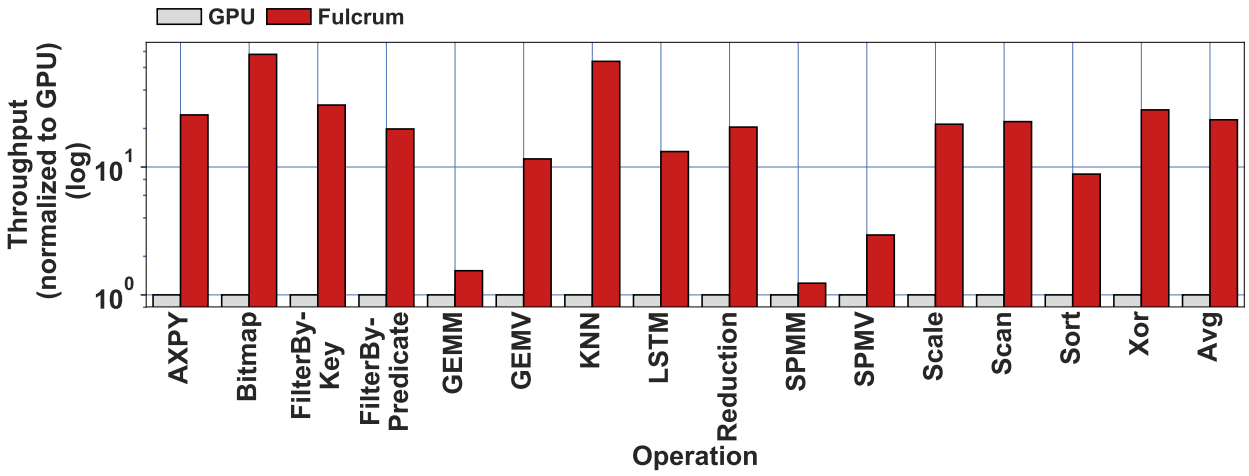


Figure 4.7: Throughput comparison against NVIDIA P100

4.5.3 Comparison against an ideal model and bitwise row-wide ALUs

The most beneficial aspect of in-situ computing is reducing the cost of data movement. Therefore we evaluated our method against an ideal model of GPU, where we only incorporated the cost of data movement between DRAM and GPU. We also evaluated our method against DRISA [2] for applications that do not have branches, where the complex operations can be emulated using bitwise operations. Figure 4.8 illustrates the throughput per memory stack of the ideal model, DRISA [2], GPU, and Fulcrum. This figure shows that Fulcrum outperforms the ideal model, on average, by 19× and up to 178.9×. However, it can not outperform the ideal model for GEMM, which has a higher locality. This figure also shows that Fulcrum can outperform DRISA [2] (the last column of Table 5.2 indicates applications that DRISA can support), often by more than two orders of magnitude. However, Fulcrum is 3.5× slower than DRISA for Xor–a bitwise task ideally suited for DRISA.

4.5.4 The effect of application characteristics

Figure 4.9 shows detailed performance metrics, collected by the NVIDIA profiler [108]. Tables 4.3 presents the definition of the metrics used in this Figure. In this Figure, the y-axis shows the normalized value of the metric, so that the y-axis value for the application with the highest metric value is one. This Figure demonstrates that applications such as Bitmap that have very high memory_read_per_computation benefit more from Fulcrum. Since reading data from DRAM is in the critical path, memory_read_per_ computation is more important than memory_write_per_computation. KNN, which gains a high speedup, has a high value of sm_inefficiency.

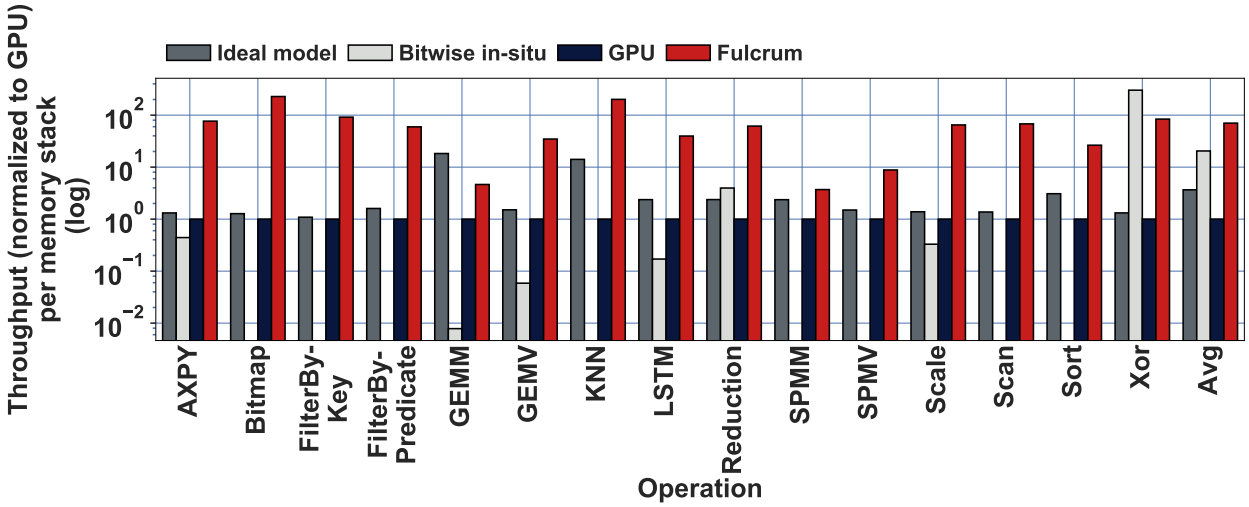


Figure 4.8: Throughput per stack comparison against the ideal model, DRISA [2], and GPU

Table 4.3: Metrics used in Figure 4.9

Metric	Definition
dram_read_bytes	Total bytes read from DRAM to L2 cache
dram_write_bytes	Total bytes written from L2 cache to DRAM.
inst_integer	Number of integer instructions executed by nonpredicated threads
inst_fp_32	Number of single-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)
stall_memory_dependency	Percentage of stalls occurring because a memory operation cannot be performed due to the required resources not being available or fully utilized, or because too many requests of a given type are outstanding
sm_efficiency	The percentage of time at least one warp is active on a multiprocessor averaged over all multiprocessors on the GPU
sm_inefficiency	1-sm_efficiency
memory_read_per_computation	dram_read_bytes/(inst_integer+ inst_fp_32)
memory_write_per_computation	dram_write_bytes/(inst_integer+ inst_fp_32)

It also shows that SPMV on GPU has the highest stall_memory_dependency, which is the result of indirect memory accesses such as x[col[j]] [122]. To implement SPMV on Fulcrum, we partition rows of the matrix among subarrays and store column indexes and non-zero values consecutively. We broadcast the index of the vector elements, followed by the corresponding value. In each subarray, the ALPU checks the column index of the non-zero value with the broadcasted index and perform multiplication and addition for matched indexes. While our implementation does not require indirect memory accesses, for highly sparse vectors, we waste many cycles for broadcasting values that will not be matched in any subarray. However, the simplified control and access mechanism and in-situ computing (which reduces the energy consumption of data movement) still provide energy benefits. Figure 4.10 illustrates that

the higher the density, the lower normalized energy-delay product (EDP) (the lower, the better). Therefore we can conclude that Fulcrum benefits applications with the density of 3-100%. Prior works have shown that many problems in statistics and sparse neural networks have such density [123].

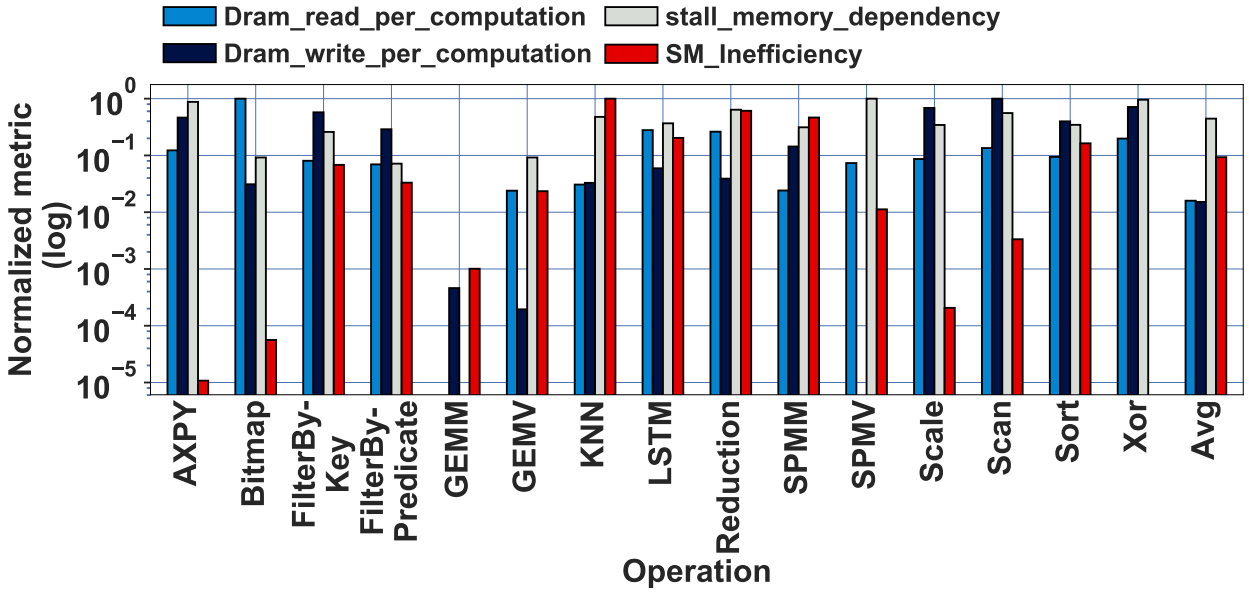


Figure 4.9: Performance metrics that affect Fulcrum's speedup

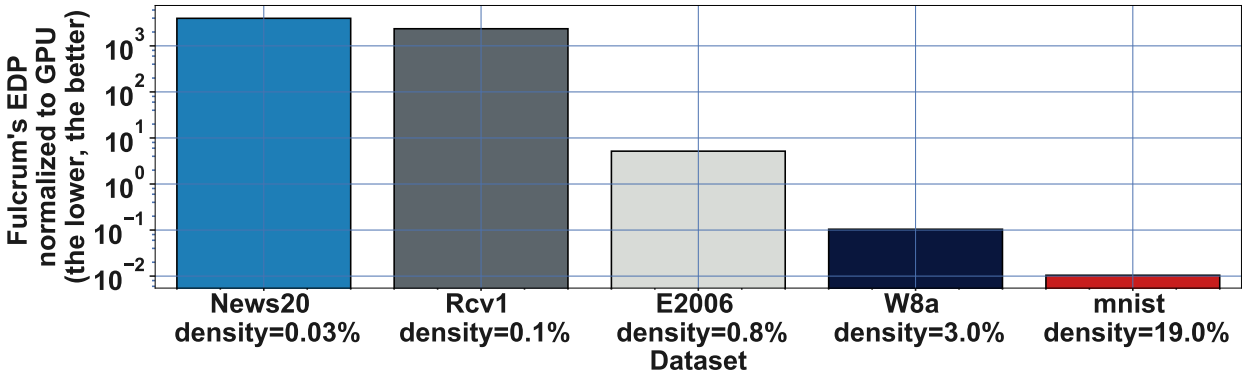


Figure 4.10: The effect of density on the EDP benefit

4.5.5 The impact of each problem

Section 4.2 lists four problems for prior in-situ approaches. In this Section, we discuss the impact of each problem.

1. Lack of flexibility: The fourth column of Table 1 demonstrates that DRISA [2], a prior in-situ approach, can not efficiently support applications such as Sort, Scan, FilterByKey, and FilterByPredicate. The hardware overhead of solutions such as a simple in-order core is 25× larger than the size of each subarray.

2. No support for complex operations: Figure 4.1 illustrates that the energy consumption of several row activations, required for emulating complex operations using bitwise operations, is higher than the energy

consumption of accessing the same data in the logic layer, nullifying the advantages of in-situ computing over in-logic layer computing.

3. Interleaving: Without reuniting interleaved bits, computation on more than 4-bit values is impractical, as bits are not physically adjacent (Section 6.3).

4. Inefficient interconnections: Figure 4.11 illustrates the overhead of copying shared values. In this figure, the y-axis shows the percentage of Fulcrum's execution time that would be spent on copying shared values in all subarrays vs. the percentage of Fulcrum's time that is spent on broadcasting. Since broadcasting is often overlapped with computation, it imposes negligible performance overhead.

Figure 4.12 shows the performance overhead of inter-subarray data movement through GDLs. This figure shows that LISA data movement can alleviate the inter-subarray data movement overhead and improve the performance of applications with inter-subarray data movement requirement such as Scan, Sort, and KNN.

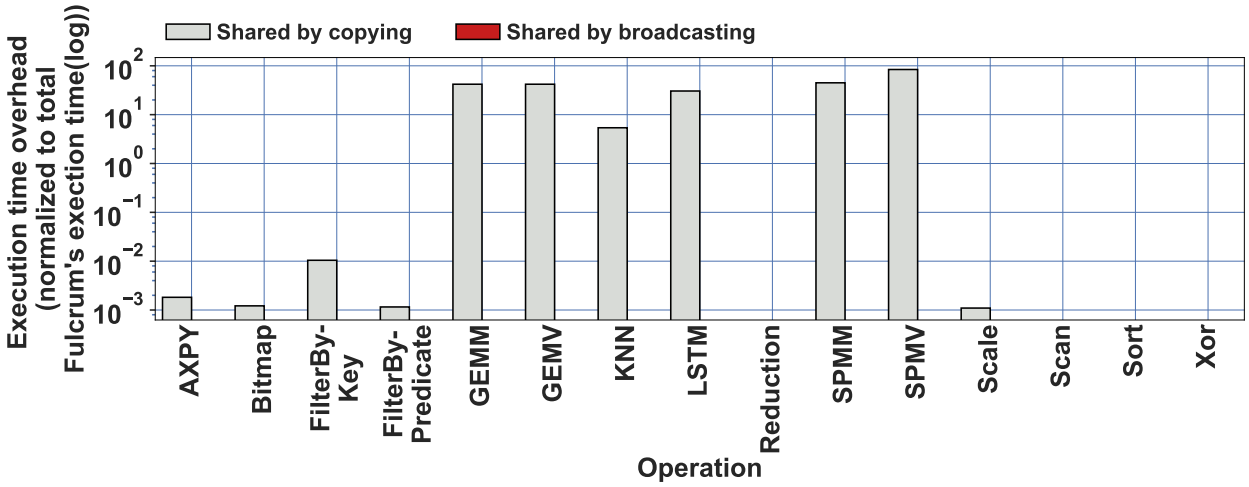


Figure 4.11: The performance overhead of copying shared values vs. the performance overhead of broadcasting. Since broadcasting and computation are often overlapped, broadcasting imposes zero/negligible overhead.

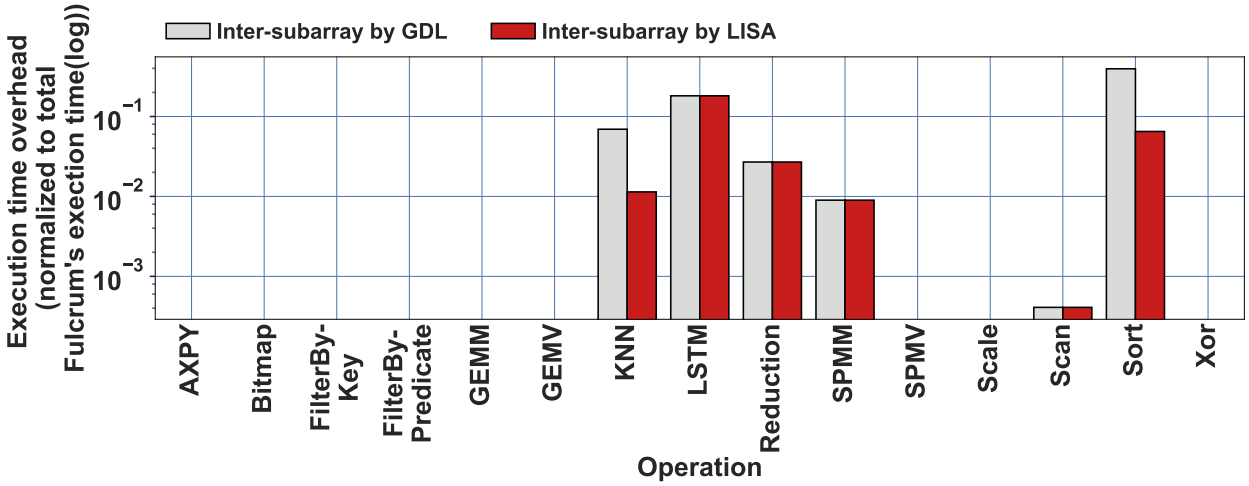


Figure 4.12: The performance overhead of inter-subarray data movement through GDL vs. LISA [3] for applications with inter-subarray data movement requirement.

Table 4.4: Area evaluation of Fulcrum

Component	Area mm^2			
	Per two subarrays		Per layer	
	Optimized	Pessimistic	Optimized	Pessimistic
Original DRAM	–	–	–	34.95
Walkers	–	0.011	–	11.26
Integer+Bitwise ALPU (4 entries)	0.0054	0.0076	5.53	7.87
Integer+Bitwise ALPU (8 entries)	0.0059	0.0083	6.09	8.51
Integer+Bitwise+Float ALPU(4 entries)	0.0088	0.0166	9.05	17.09
Integer+Bitwise+Float ALPU(8 entries)	0.0093	0.0173	9.52	17.73

4.5.6 Area evaluation

A high capacity accelerator is desirable as it can support applications with large footprints. Therefore we targeted an 8GB accelerator with eight layers (1GB per layer). Fulcrum has two major types of components added to the commodity DRAM: *(i)* Walkers and *(ii)* ALPUs. Table 5.5 lists the optimistic and pessimistic area evaluation of these components with different configurations. Our evaluation shows that an 8GB integer Fulcrum, with 4 entries of the instruction buffer, is achievable by eight layers, where the area of each layer optimistically (pessimistically) is $51.74mm^2$ ($54mm^2$). A 4-entry, 8GB float Fulcrum is achievable by eight layers, where the area of each layer optimistically (pessimistically) is $55.26mm^2$ ($63.3mm^2$).

4.5.7 Energy consumption

Figure 4.13 compares the energy consumption of two configurations of Fulcrum: (i) integer Fulcrum, and (ii) float Fulcrum to GPU. This Figure illustrates the energy consumption spent (both dynamic and static) on three parts: (i) data movement (on on-chip and off-chip memory elements and interconnections), (ii) control (instruction fetch units and instruction schedulers), and (iii) computation (ALUs and FPUs). Fulcrum reduces the total energy consumption, compared to GPU, on average by 96%. Our evaluation shows that float Fulcrum reduces the energy consumption of movement, control, and computation by 97%, 73%, and 24%, respectively. Unlike the energy reduction in control and data movement (which are expected), the energy reduction in computation is unexpected as Fulcrum uses larger technology size than GPU for computation. Our evaluation shows that the dominant factor in energy reduction, for memory-intensive applications (where the computation units are often waiting for the data), is the reduced execution time, which reduces the static power consumption. For computation-intensive applications such as GEMM, Fulcrum increases the computation energy by 509%, as expected.

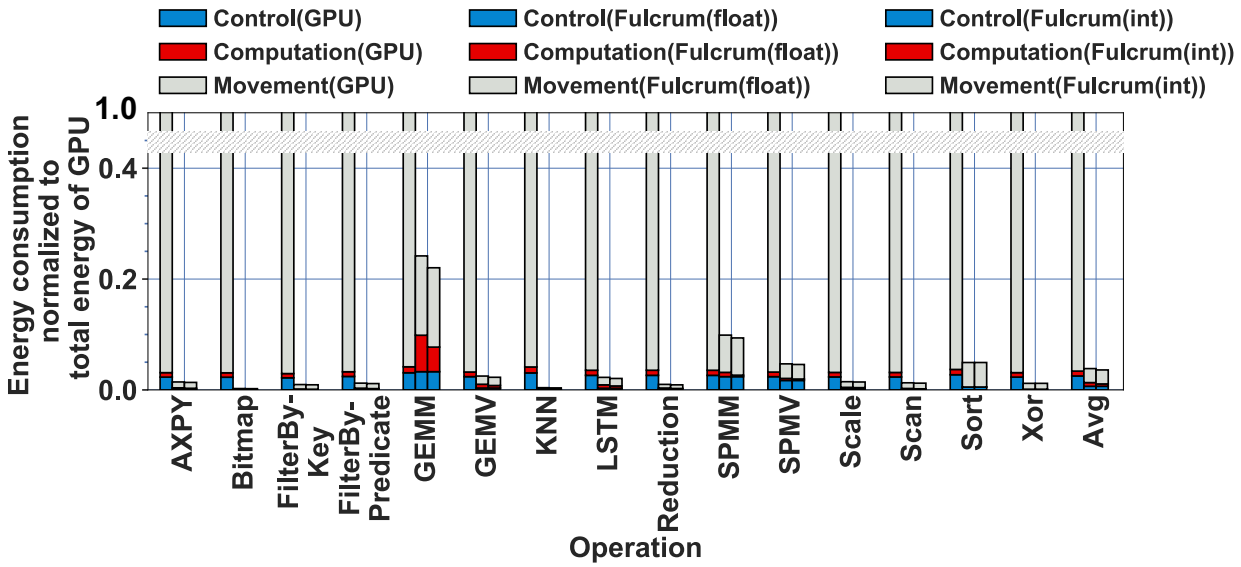


Figure 4.13: Breakdown of energy consumption

4.5.8 Power evaluation

Figure 4.14 compares the power consumption of GPU, float Fulcrum, and integer Fulcrum. This figure illustrates that, on average, float Fulcrum (integer Fulcrum) decreases the power consumption by 72.3% (73.8%). Our detailed evaluation shows that 33.2% percent of the power consumption of float Fulcrum is spent on row activation, 34.71% is spent on moving data to the side of the subarray, 13.1% is spent on computation, and 14.6% is spent on control. The rest is spent other forms of data movement such as broadcasting, LISA movement, and collecting the partial results.

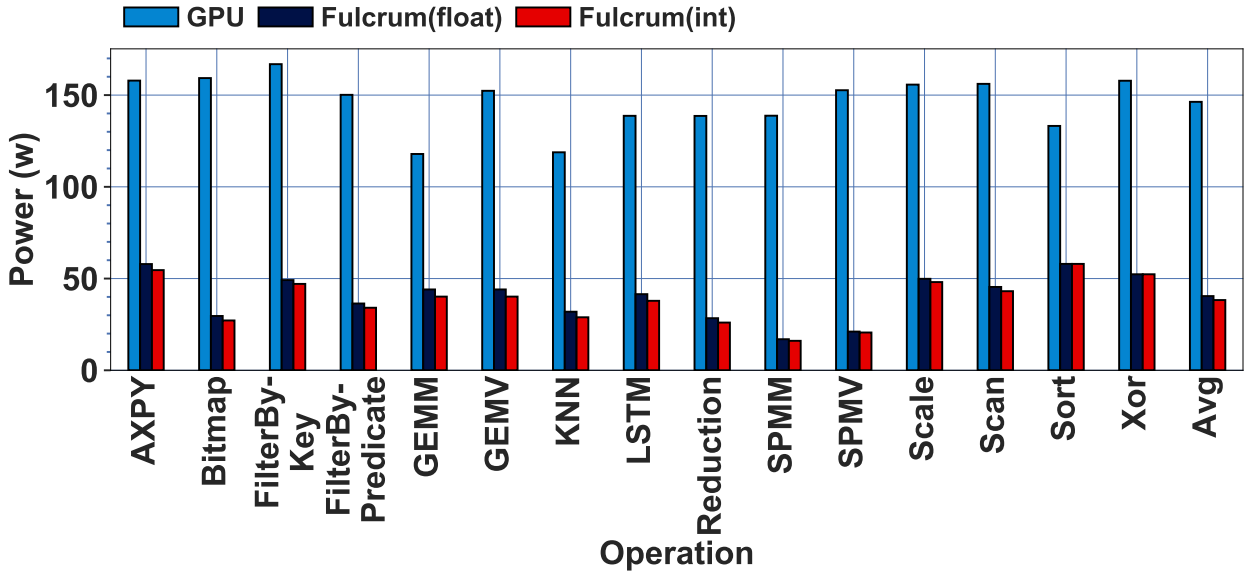


Figure 4.14: Power consumption of integer Fulcrum and float Fulcrum vs. GPU

4.5.9 Performance under power budget

The power budget of any accelerator directly affects the choice of interface and cooling system. Prior works [94, 84] suggest that a power budget of 10 Watts is practical through current 3D-stacked memory interfaces. With a higher power budget, deployment as a PCIe/CXL peripheral is required to deliver the required power for the accelerator and more complex and, consequently more expensive cooling system is required. To change the power consumption of Fulcrum, we can simply change the frequency (however, we do not increase the frequency beyond 164 MHz, which we treat as our maximum frequency). Figure 4.15 illustrates the throughput of Fulcrum (normalized to GPU) under three power budgets: 10, 40, and 60 Watts. This Figure shows that even with the power budget of 10 watts, Fulcrum (one stack) outperforms a high-performance GPU (with three stacks of memory) by 6× on average, and up to 25×. However, under this power budget, Fulcrum slows down GEMM and SPMM.

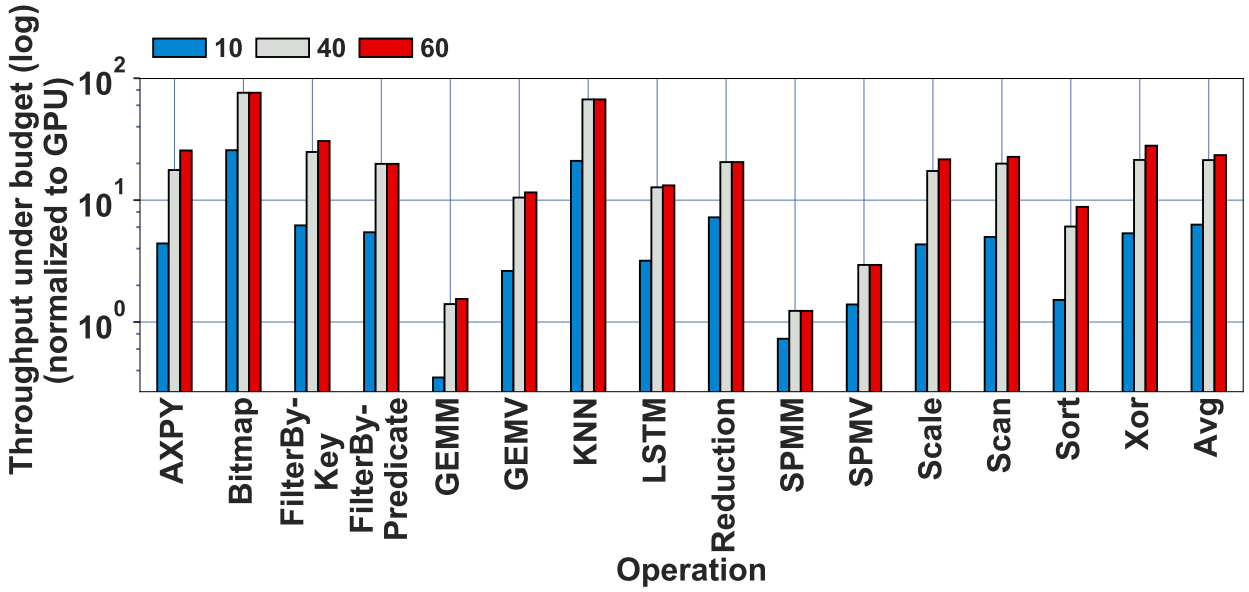


Figure 4.15: Performance under different power budgets

4.6 Related Work

Traditional solutions for reducing the cost of data movement have limited benefits. Prefetching techniques can not alleviate the energy cost of data movements. Forwarding on-chip blocks techniques [47] are only applicable when cores share values. New techniques such as quantized memory hierarchies [18] mostly benefit applications that can tolerate errors. Therefore, in this section, we discuss only prior works with processing units near memory elements. We categorize these works into three groups. The first group only supports bitwise operations. The second group uses the analog computing capability of memory cells of some of the

NVM technologies such as multi-bit memristors. Finally, the third group places flexible cores in the DRAM layer. This Section discusses how Fulcrum differs from such approaches.

In-situ computing with bitwise operations

SRAM-based [22] and DRAM-based [23, 2, 12, 24] in-situ accelerators often support only bitwise operations. An NVM-based in-situ accelerator, Pinatubo [25], also proposes to change the sense amplifiers to enable bitwise operation for NVMs. A limited number of applications can benefit from bitwise operation. Recent works [2, 26] implemented binarized and 2-bit quantized deep neural networks using in-situ accelerators. In most memory technologies, employing row-wide complex ALUs imposes a significant hardware overhead. Li et al., [2] evaluated the overhead of adding a 4-bit adder per every four bits of the row buffer and concluded that it imposes more than 100% area overhead. Our proposed method has two advantages over these works. First, we enable 32-bit complex operations such as multiplication and addition without imposing a substantial hardware overhead. Second, our proposed method increases the flexibility of accelerators to support a wider range of applications.

Complex operations using analog computation capability of memory cells

Several prior works [27, 28, 29] employed the analog computation capability of ReRAM technology to perform matrix-vector multiplication. Our proposed method differs from such methods in three aspects. First, these technologies are memory-technology dependent and often use multi-bit memristor devices, which are unreliable. These techniques are neither applicable to SRAM/DRAM, nor commercialized NVM memories [30] such as 3DXpoint [31]. Second, they require ADC/DAC blocks that impose significant hardware overhead (98% of the total area) and power overhead (89% of the total power consumption) [30]). Third, performing multiplication and addition operations by approximately measuring the current, introduces potential imprecision. A recent work, FloatPIM [30] designs a CNN accelerator for training by enabling floating-point matrix-vector multiplication in memory blocks, without requiring ADC/DAC. To this end, this approach copies the vector (which is the shared value), not only in each subarray but per each row of the matrix, to enable parallel multiplication and addition, imposing capacity and energy overhead. Furthermore, it implements addition and multiplication using multiple bitwise operations and depends on the computation capability of memristors. More importantly, they change the entire memory architecture and interconnection, and as a result, the memory cannot be efficiently be accessed as a normal memory.

Flexible cores in DRAM layers

In the first round of research on processing in memory, in the 1990s, a variety of works [33, 34, 35, 36, 37, 38]) proposed to add flexible cores per each bank or per entire DRAM chip. Some of these proposals [39] add one processor per entire DRAM, plus multiple bank-level buffers. In these proposals, each column of these buffers moves toward the single processor. This imposes a high cost of data movement, is not scalable for modern

DRAMs with many banks and subarrays, and limits parallelism. Our buffers enable sequential word-level access with subarray-level parallelism. Recently UPMEM [40] has described a product with a complex core per each bank of 64 MB. Fulcrum instead adds a simple processing unit per 1 MB and consequently provides higher parallelism and imposes less overhead for data movement (Our evaluations show that the energy consumption of accessing data at the edge of a bank through GDLs is at least 3× as much as that of a floating-point addition.). More importantly, as Section 4.3.1 explains, traditional cores impose a significant overhead of control and access for sequential operations. This overhead might be acceptable for far-memory cores for two reasons. First, for these cores, the ratio of this overhead compared to the overhead of data movement is negligible. Second, these cores use the small technology size and have a high frequency, whereas cores in DRAM layers have to use large technology size and have a low frequency. Consequently, the overhead of the extra cycles for such control and access mechanism becomes significant for in-memory cores. Fulcrum accelerates sequential operations and reduces the overhead of access and control.

4.7 Conclusion and Future works

For memory-intensive tasks, data movement dominates computation. Keeping computations close to the subarray's row buffer avoids these data-movement overheads, while simultaneously enabling high throughput, thanks to subarray-level parallelism. Fulcrum overcomes key limitations of prior in-situ architectures, by placing a scalar, full-word processing unit at the edge of each pair of subarrays. We show that sequentially processing a row (instead of bit-parallel processing of the entire row buffer) with full-word computation ability allows a much wider range of tasks to leverage in-situ processing, such as a full range of arithmetic operations, key sparse and dense linear algebra tasks, operations with data dependencies, operations based on a predicate, scans and reductions, and so forth. This significantly broadens the market for an accelerator for memory-intensive processing. Leveraging DRAM technology as the basis for in-situ processing also enables high total data capacity and high total parallelism, thanks to the large number of subarrays. Our proposed method provides, one average (up to), 70 (228)× speedup per memory stack over a server-class GPU.

So far we discussed the challenges regarding implementing our method for DRAM. However we believe the same simplified control and access mechanisem can be employed for SRAM-based and NVM-based accelerators.

Fulcrum with SRAM-based memory technologies can benefit from the high frequency and more efficient logic of SRAM technologies. Recent works have utilized SRAM for in-situ pattern matching and have shown at least two orders of magnitude higher throughput per unit area than a prior in-DRAM solution [124, 125, 126, 127, 128, 129]. More importantly, due to the flexibility of SRAM-based designs, we can include

memory elements with efficient random access, which can support a broader range of applications. However, due to the lower capacity of SRAM, the number of subarrays are lower than DRAM, limiting parallelism. Furthermore, the current structure of SRAM-based memories is less suitable for Fulcrum as the row buffers are shorter. Therefore, a future work can optimize SRAM structures for employing Fulcrum.

NVMs have a higher capacity and higher number of subarrays and, hence, it can provide higher parallelism. Current NVM-based in-situ approaches use the computation capability of memory cells. This type of in-situ computing has three problems: (i) the ADC and DAC impose a significant hardware overhead, (ii) it is not flexible, and (iii) it introduces error, and the error depends on the number of rows in each subarray, limiting the size of each subarray and limiting the capacity. A future work can evaluate Fulcrum for NVM and investigate the challenges such as low endurance.

Chapter 5

Gearbox: A Case for Supporting Accumulation Dispatching and Hybrid Partitioning in PIM-based Accelerators

5.1 Introduction

In current computing systems, the latency and energy consumption of fetching data from off-chip memory can be 2-3 orders of magnitude higher than an arithmetic operation [4]. Processing-in-memory (PIM) architectures alleviate this data movement overhead by placing processing units near memory segments (banks or subarrays) [130, 2, 131, 13].

SpMV and SpMSpV are essential computational kernels that are widely used and *memory-intensive* (requiring few computations per loaded datum from memory). The generalized forms of SpMV and SpMSpV, where the multiplication and addition can be replaced by other operations, appear in many important application domains such as machine learning (e.g., Support Vector Machine and Sparse K-Nearest Neighbor) and graph processing (e.g., Page Rank) [132, 133, 134]. Due to SpMV and SpMSpV kernels' memory-bound nature and widespread applications in various domains, they are natural candidates for PIM acceleration. Adding support for these kernels to PIM-based accelerators can boost such applications' performance, expand the market for PIM, and increase vendors' motivation in PIM investment.

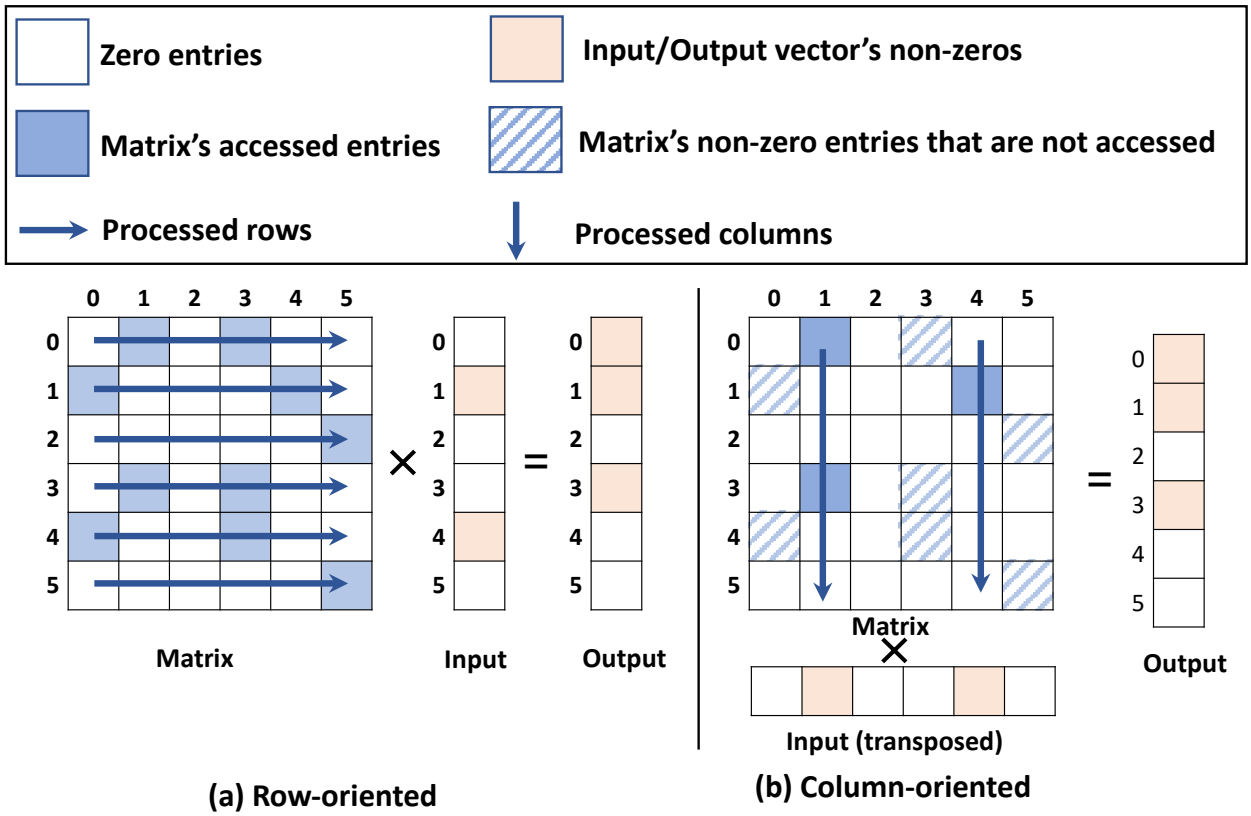


Figure 5.1: The row-oriented approach processes all rows, whereas the column-oriented approach processes only the columns corresponding to the non-zero entries of the input vector. In (b), the input vector is transposed to illustrate the relation between non-zero entries of the input vector and the processed (activated) columns of the matrix.

However, existing PIM architectures often are only optimized for regular kernels by providing high parallelism using SIMD units [131, 135] or bit-level parallelism [130, 2]. In this paper, we introduce a PIM architecture that provides high parallelism for SpMV and SpMSpV. Later, we demonstrate that our proposed architecture outperforms SIMD approaches for regular kernels as well.

There are two major approaches for SpMV and SpMSpV: i) row-oriented or matrix-driven approach (Figure 5.1(a)), and ii) column-oriented or vector-driven approach (Figure 5.1(b)). The row-oriented approach requires processing *every* non-zero element of the input matrix for both SpMV and SpMSpV. On the other hand, for SpMSpV, the column-oriented approach processes only the columns corresponding to the non-zero entries of the input vector. We refer to these columns and their non-zero entries as activated columns and activated entries. As a result, the column-oriented approach is more efficient for SpMSpV [136].

While the column-oriented approach is common in GPU, CPU, FPGA, and ASIC solutions for SpMSpV [137, 138, 139, 140, 141, 142, 143, 144, 145, 134, 133], none of the prior bank-level or subarray-level PIM-based SpMV accelerators [146, 132] have implemented column-oriented processing. To maximize the

benefits of column-oriented processing, we need to address two issues: i) random accesses to remote memory segments and ii) *power-law* column length distribution.

Random accesses to remote memory segments:

Processing SpMV and SpMSpV in PIM requires the compressed matrix, the input vector, and the output vector to be partitioned among memory segments. With both row-oriented and column-oriented approaches, the processing units near each segment require access to data that is stored in another memory segment. For example, in Figure 5.2 (a), one of the multiplication and addition required for generating $Output[3]$ is $Output[3] += Matrix[3,0] * Input[0]$. However, Figure 5.2 (a) shows that $Input[0]$ and $Matrix[3,0]$ reside in Subarray 1 (S1), but $Output[3]$ resides in Subarray 2 (S2). Therefore, the processing unit in S1 does the multiplication part ($Input[0] * Matrix[3,0]$) locally but has to access Subarray 2 (S2) to write the result of multiplication in $Output[3]$.

The remote write accesses are remote accumulations that do not require any mechanism for enforcing the order of operations. Therefore, the result of multiplications can be sent to be accumulated in the destination memory segment. For example, S1 can send the multiplication result to S2 to be added to $Output[3]$ in S2 and continue processing another multiplication and do not need to wait until the accessed operand arrives from a remote memory segment.

Accordingly, we propose *Accumulation dispatching*. In this mechanism, one dedicated subarray in every bank acts as a dispatcher for remote accumulations. Without the dispatcher, each remote accumulation would interrupt the normal processing of the processing unit in the remote subarray. The dispatcher collects all the remote accumulations and sends them to their destination once the destination subarray's processing ends. This solution sacrifices only 6% of capacity. In Section 5.7.3, we evaluate an alternative albeit impractical approach.

Power-law column length distribution:

Real-world sparse matrices' column lengths follow the *power-law* distribution [147]. That means most of the rows/columns contain very few non-zero entries (referred to as *short* rows/columns), while the remaining row/columns have orders of magnitude higher numbers of non-zero entries (referred to as *long* rows/columns). The natural way of partitioning a matrix for the column-oriented approach is to assign a few full columns to each memory segment, where the input entries that activate these columns reside. However, with a *power-law* column length distribution, whenever a long column gets activated, the processing unit of the subarray that has this column has to perform many more multiplications than other processing units, causing load imbalance. We also observed that, with naive column-oriented partitioning, most of the remote accumulations are due to long columns.

To address these issues, we propose *Hybrid partitioning* scheme that treats short and long columns differently. We partition the short columns in a normal column-oriented way. The long columns' non-zero entries are distributed among all memory segments, so that each non-zero entry and its corresponding entry in the output vector reside in the same memory segment. We also propose hardware support for our proposed partitioning technique. To lower the overhead of our hardware support, we reorder the matrix so that the long columns/rows are the first columns/rows of the matrix and their index is lower than a threshold. As a result, distinguishing the indexes corresponding to these long columns and long rows can be implemented using a comparator and a latch that holds the threshold.

Figure 5.2 (b) shows that with Hybrid partitioning, the long column no longer causes any remote accumulation, since $Matrix[3:5,0]$ and $Output[3:5]$ reside in the same subarray. This partitioning also alleviates load imbalance, because all processing units co-operate on processing an activated long column.

With Hybrid partitioning, for multiplications, all subarrays need to access the input vector entries that activate long columns. We place these entries in the logic layer (one of the layers in 3D stack memories, introduced in Section 6.2) and broadcast them to all subarrays. For example, in Figure 5.2 (b), we place $Input[0]$ in the logic layer.

Based on these two key ideas, we propose *Gearbox*, which adds efficient hardware supports for column-oriented processing to PIM-based accelerators. We use Fulcrum [13] as the baseline PIM architecture for Gearbox. Fulcrum places one lightweight single-word processing unit at every two subarrays to achieve high parallelism. The subarray-level single-word processing allows parallel and independent access per single-word ALU. Therefore, unlike SIMD approaches, the ALUs do not have to wait for all the operands to be collected. However, Fulcrum [13] only supports sequential accesses. It does not support local random accesses (i.e., random access within the same subarray) and remote accesses required by the SpMV and SpMSpV kernels. We modify Fulcrum to add support for a new important range of applications by enabling local random accesses, as well as adding support for our proposed Accumulation dispatching and Hybrid partitioning. Our support for local random accesses, Accumulation dispatching, and Hybrid partitioning is programmable, enabling future works to map more irregular kernels to our proposed architecture.

Our proposed method, Gearbox, with just *one* memory stack, delivers on average (up to) 15.73× (52×) speedup over a server-class GPU, NVIDIA P100, with *three* stacks of HBM2 memory. Note that the P100 is not the state-of-the-art GPU and newer GPUs have even more memory stacks. Compared to GPUs with more memory stacks, Gearbox remains highly competitive in terms of speedup per stack because Gearbox delivers on average 45× speedup per stack compared to NVIDIA P100.

Gearbox also outperforms an ideal model of SpaceA [146], a PIM-based SpMV accelerator that only supports row-oriented processing (assuming no area overhead, perfect load balancing, and no penalty for remote reads for SpaceA) by 58× (447×) per area.

Our paper makes the following contributions:

- Proposing a highly parallel architecture that can exploit the parallelism for regular kernels, as well as SpMV and SpMSpV.
- Proposing the first in-memory-layer approach (near banks/subarrays) that implements column-oriented processing, which is more efficient than row-oriented processing.
- Proposing Hybrid partitioning to reduce remote accumulations and alleviate load balancing.
- Proposing hardware support for remote accumulations and Hybrid partitioning.

5.2 Background

5.2.1 HMC-like configuration vs HBM-like configuration

HMC has short rows (e.g., 256 bytes) and narrow columns (e.g., 32-bit) and GDLs, whereas HBM2E [148] has wide rows (e.g., 1 KB) and wide columns and GDLs (e.g., 256 bit).

We choose memory configurations with short rows (e.g., 2048 bits), such as HMC [19, 13], because memory configurations with short rows are more efficient for parallel row activations and random accesses, where only a few words of a row are useful.

5.2.2 Sparse operations

We denote generalized matrix-vector multiplication as $Output[:] = Matrix[:,:] \times Input[:]$, where $Input[:]$ and $Output[:]$ are vectors, and $Matrix[:,:]$ is a matrix. By "generalized", we mean multiplications and accumulations can be replaced by any other operation with similar properties (e.g., commutativity). In most applications, we need an extra step on the output vector $finalOutput[:] = Output[:] + \alpha y[:]$, where α is a scalar value and $y[:]$ is a vector. The addition and multiplication in this step can also be replaced by any other operation. We refer to this step as Applying.

Many applications can be formulated as SpMV and SpMSpV [132, 134, 133]. For example, Single-Source Shortest Paths (SSSP), a graph processing application, can be formulated as SpMSpV, in which multiplication is replaced by addition, and the accumulation operation is replaced by minimization.

5.2.3 Sparse matrix representations

There are two main data representations for sparse matrices: (i) compressed sparse rows (CSR) and (ii) compressed sparse columns (CSC). CSC/CSR stores the matrix in three arrays containing: (i) non-zero values ($Values$), (ii) row/column indices of non-zero values ($Indexes$), and (iii) offsets ($Offsets$) that refer to the positions of the start of the columns/rows in both $Values$ and $Indexes$ arrays.

CSC representation is more efficient for column-oriented processing, as it has the position of the start of each column. We can pair the $Values$ and $Indexes$ arrays into one array (CSC_Pair), as shown in Figure 5.3.

5.3 Motivation and key ideas

5.3.1 Support for column-oriented processing using accumulation dispatching

Figure 5.5 shows that the column-oriented algorithm only processes the columns that correspond to non-zero entries of the input vector. Therefore, column-oriented processing operates on the sparse format of the input vector (lines 4-5). We refer to this format of the input vector as the frontier (line 5, currFrontier in Figure 5.5). Column-oriented processing also requires random access to the output vector (lines 20-21). When we partition the matrix and the input/output vectors among memory segments, the accumulation in line 21 can be *remote* or *local*. For example, in Figure 5.5, consider a subarray containing $Matrix[:, j:k]$, $Input[j:k]$, and $Output[j:k]$. In line 21, if $j \leq row\_index \leq k$, the accumulation is a local accumulation. Otherwise, it is a remote accumulation.

Key idea 1: Accordingly, we add hardware support for distinguishing remote accumulations from local accumulations by placing a comparator and two latches that hold the range of index of local accumulations. We also propose a mechanism for dispatching remote accumulations, *Accumulation dispatching*. In this mechanism, one specialized subarray in every bank acts as a dispatcher for the remote accumulations. We elaborate on the details of this mechanism in Section 5.4.

5.3.2 Reducing remote accumulations and balancing the load by supporting Hybrid partitioning

Figure 5.4 shows the column length distribution of our evaluated datasets, where the x-axis (log scale) shows the column length and the y-axis is the percentage of columns within that range. This figure demonstrates that there are only a few long columns, but they are orders of magnitude longer than the other columns. Same goes for the long rows. We refer to the top $X\%$ (e.g., 0.01%) of columns/rows as long columns/rows. This threshold is configurable in our architecture.

Figure 5.2 (a) in Section 6.1 demonstrates that, with column-oriented partitioning, where each subarray has a few full columns, the long columns can cause many remote accumulations and significant load imbalance among processing units.

Key idea 2: Given these observations, to both balance the load and reduce the number of remote accumulations, we propose Hybrid partitioning. Figure 5.2 (b), in Section 6.1, illustrates that Hybrid partitioning treats short and long columns differently. We partition the short columns in a column-oriented way but divide the long columns among all subarrays. Consequently, each part of a long column resides in the same subarray in which its corresponding part of the output vector resides, eliminating remote accumulations. Furthermore, all subarrays cooperate for processing long columns, alleviating the load imbalance.

In iterative algorithms, the output vector becomes the input vector of the next iteration. Therefore, in the next iteration, all subarrays for multiplication require accessing the output vector entries that activate a long column. We place the output vector entries corresponding to long columns in the logic layer. In the subsequent iterations, they are broadcast to all subarrays from the logic layer, eliminating the need of copying from the output vector to the input vector. Since there are only a few activated long columns in each iteration, the broadcasting imposes negligible overhead. The overhead is evaluated in Section 5.7.4.

Real-word matrices may also contain a few long rows. Figure 5.6 (a) shows that these long rows can trigger many remote accumulations. To reduce this remote accumulation overhead, we place the output entries corresponding to the long rows in the logic layer. The logic layer provides more efficient random accesses, since it has SRAMs.

To implement Hybrid partitioning, our subarray-level processing units should be able to distinguish among input/output entries corresponding to the long columns. We reorder the matrix so that the long columns/rows of the matrix and their index are lower than a threshold. As a result, we can implement this distinction by using a comparator and a latch that keeps the index of the last long column/row. Section 5.6 explains that this one-time cost is acceptable [147, 149, 146, 150, 151].

To further minimize the overhead of accumulation of long columns/rows, we added an optional optimization, where we replicate the output vectors corresponding to the long columns/rows in all subarrays. Then we accumulate the long rows, first locally in each subarray and then in the logic layer (Figure 5.6 (b)). If we choose 0.01% of rows/column as long rows/columns, the capacity overhead of this technique stays below 1.7%.

5.4 Proposed Architecture

We use Fulcrum[13] as the baseline PIM-based architecture. Motivated by characteristics of memory-intensive applications, where there are few simple operations per loaded datum from memory, Fulcrum places one simplified sequential processing unit per pair of subarrays. Each subarray-level processing unit (SPU) has a few registers, an 8-entry instruction buffer, a controller, and an ALU. In Fulcrum, every pair of subarray has three row-wide buffers, called Walkers. The Walkers load an entire row from the subarray at once, but the processing units sequentially access and process one word at a time. The sequential access is enabled by using a one-hot-encoded value, where the set bit in this value selects the accessed word. Therefore, to sequentially process the row, the processing unit only needs to shift the one-hot encoded value.

We chose Fulcrum because it is more flexible and more efficient than bank-level SIMD approaches for three reasons. First, the three Walkers enable three concurrent sequential accesses. Second, Fulcrum can exploit the parallelism for operations with data dependency because Fulcrum processes row-wide buffers sequentially. Third, Fulcrum can efficiently exploit the parallelism for operations with branches because each subarray has an 8-entry instruction buffer that allows each ALU to perform a different operation independently.

However, Fulcrum can only support sequential accesses and is inefficient for irregular kernels that require random accesses, communications among subarrays, or load balancing. In this paper, we: (i) modify the sequential access mechanism of Fulcrum to enable local random accesses, (ii) add in-memory-layer interconnection and a dispatching mechanism to enable remote accumulations, (iii) add ISA and hardware support for our proposed Hybrid partitioning, which minimizes communications among subarrays and provide hardware support for load balancing. Our modifications add only 10.93% area overhead to Fulcrum but enable exploiting the high parallelism of Fulcrum for a new range of important applications.

Figure 6.2 illustrates our proposed architecture, which is based on 3D-stacked memories. Similar to prior works [84, 13], every vault has a simple in-order core with a 32KB SRAM scratchpad underneath it, in the logic layer. A ring interconnection topology (Figure 6.2 (a)) connects the banks in each memory layer. Subarrays within a bank are connected through a line interconnection topology (Figure 6.2 (b)).

As shown in Figure 6.2 (b), we have two types of SPUs. Subarrays closest to the ring interconnect contain *Dispatcher SPUs*. Other subarrays contain *Compute SPUs*.

The logic layer components launch a kernel (or one step of a kernel) by broadcasting at most 8 instructions to all Compute and Dispatcher SPUs and loading new values from each subarray to the associated latches. Our launching mechanism is similar to Samsung's PIM [131] and poses negligible overhead compared to the total execution time.

In this section, we elaborate on the role of each part of our architecture, using a simple kernel, $C[A[:]] += B[:]$. At a high level, a Compute SPU reads the i^{th} entry of array $A[:]$, compares this entry against three latches, and processes the accumulation differently based on the result of this comparison. These three latches are $FirstLocal3$, $LastLocal3$, and $LastLong3$. If $FirstLocal3 \leq A[i] \leq LastLocal3$, the accumulation is a local accumulation. If $0 \leq A[i] \leq LastLong3$, the accumulation is again a local accumulation but on the replicated part, $C[0 : LastLong3]$. Otherwise, the accumulation is a remote accumulation. In this case, the Compute SPU sends the index-value pair ($A[i]$ and $B[i]$) to the Dispatcher.

We use this simple example to introduce our modifications to Walkers, provide a walk-through example, and explain the role of Dispatchers. In the end, we elaborate on the details of the instruction format.

5.4.1 Walkers and indirect accesses

PIM targets memory-intensive applications that process large arrays. In our architecture, each Walker read from or write to one of these large arrays. The Start1/2/3, shown in Figure 6.2 (c), determine the row address. The End1/2/3 latches determine the end address of the arrays associated with Walker1/2/3, respectively.

For example, Walker1 loads one row from $A[:]$. Then, the controller accesses the row one word at a time by shifting the one-hot-encoded value of Walker1. Once the set bit in the one-hot-encoded value reaches the last position, the controller loads a new row from array $A[:]$.

In our previous example, however, the array $C[:]$ was being accessed randomly using $A[:]$'s entries. When an array is the index of another array, the access is called an indirect access. To enable indirect accesses, we add two fields to the instruction format that determine which register contains the index of the indirect access and which Walker should be used for loading the row containing the accessed word. Our controller derives the row address and column address using the index. To select the accessed word from the row, we shift the one-hot-encoded value and increment a counter until the counter equals the column address. To hide the overhead of shifting, we overlap loading a new row into the Walker and shifting the one-hot-encoded value and use the sub-clock, introduced in [13]. This simple modification enables parallel and independent random access per ALU in the accelerator, enabling applications with high access divergence.

5.4.2 A walk-through example

Figure 5.8 illustrates a walk-through example of how our architecture processes the first step of $C[A[i]] += B[i]$, using four instructions. The Compute SPUs iterate over these instructions for each entry of $A[:]$ and $B[:]$.

Figure 5.8 ❶ shows the pseudo format of instruction[0], and Figure 5.8 ❷ illustrates the operation performed by instruction[0]. With this instruction, the Compute SPUs load one word from Walker1 into $Walker1Reg$ and one word from Walker2 into $Walker2Reg$.

With instruction[1], as shown in Figures 5.8 ❸ and ❹, the SPU moves $Walker2Reg$ to $reg1$ and compares the $Walker1Reg$ against the three latches ($FirstLocal3$, $LastLocal3$, and $LastLong3$). If $FirstLocal3 \leq Walker1Reg \leq LastLocal3$, the Compute SPU derives the row address and column address of $C[Walker2Reg]$, using Start3 latch. If $0 \leq Walker1Reg \leq LastLong3$, the row address is derived using the $LongStart3$ latch that stores the start of the replicated part of $C[:]$ (i.e., $C[0 : LastLong3]$). Using the indirect mechanism that we explained in the previous subsection, SPU loads $C[Walker1Reg]$ into $Walker3Reg$. If the accumulation is a remote accumulation, the controller places the index and the value stored in Reg1 and Walker1Reg on the line interconnection's port (DownPort in Figure 6.2 (c)) and returns to instruction[0].

Otherwise, Instruction[2], as shown in Figures 5.8 ❺ and ❻, performs the accumulation ($Walker3Reg+ = Reg1$).

Instruction[3], as shown in Figure 5.8 ❼, writes the $Walker3Reg$ register to the Walker3, loads one word from Walker1 into $Walker1Reg$, loads one word from Walker2 into $Walker2Reg$, and returns to Instruction[1]. The controller iterates over these instructions until all A[i] entries are processed.

5.4.3 Dispatcher and the bank-level switch

The Dispatcher SPUs are located in the subarrays closest to the ring interconnect (Figure 6.2 (c)). They are primarily responsible for routing remote accumulation packets. To assist in packet forwarding, the Dispatcher SPUs contain a switch that keeps the range of the indexes assigned to its bank and its layer in corresponding latches.

In our previous example, the Compute SPUs send any non-local index-value pairs to the Dispatcher in the bank. When the Dispatcher receives an index-value pair, if the index belongs to its bank, the Dispatcher loads the index-value pair in one of its walkers. If the index-value pair belongs to the same memory layer, the Dispatcher places it on the ring interconnection's port. Otherwise, the Dispatcher forwards the index-value pair to a different memory layer via TSVs. As a result, multiplications and local accumulations are overlapped with sending remote accumulations.

After the multiplication and local accumulation, to complete the remote accumulations, we need two additional steps. In the first step, the Dispatchers start sending the index-value pairs to Compute SPUs in the same bank. In the second step, each Compute SPU processes the received index-value pairs to perform the final accumulation (using instructions that are very similar to the instructions in the first step).

5.5 SpMSpV walk-through

We can map SpMSpV to our architecture using the following steps.

Step1 (FrontierDistribution): In Section 6.2, we explained that the sparse format of the input vector is called the frontier. In the first iteration, we partition and distribute the frontier among subarrays. In most algorithms, the first frontier is very small (e.g., one entry for BFS). In iterative applications, the frontier is generated in previous iterations and already resides in subarrays in which their corresponding columns reside, except for the output entries that correspond to long row/columns, which reside in the logic layer. At the start of each iteration, we broadcast the entries residing in the logic layer to all subarrays and append them to the frontier array in each subarray.

Step2 (OffsetPacking): This step packs the column offset, column length, and the values from the frontier array that should be multiplied in the column into a new array. Figure 5.9 shows the pseudo-code of this step.

Step3 (LocalAccumulations): This step multiplies each value of the frontier with its corresponding column. Figure 5.10 demonstrates the pseudo-code of this step. In this step, if a clean value is being updated, the clean value indicator and its row index will be sent to the Dispatcher.

Step4 (Dispatching): In this step, the Dispatcher sends all the stored entries (index-value pairs) to their destination subarrays. Here, the Dispatcher's Walker acts as a buffer.

Step5 (RemoteAccumulations): In this step, the SPU sequentially processes index-value pairs received in the previous step and performs the accumulations. Also, in this step, if the value in the index-value pair is a clean-value indicator, the index of clean-value is appended to the corresponding array.

step6 (Applying): This step processes the array containing the non-zero indexes to generate the frontier for the next iteration, initializes the output vector to clean indicators, and sends long-activating entries to the logic layer to be reduced and applied there. It also performs the apply operation ($finalOutput[:] = Output[:] + \alpha y[:]$, which is explained in Section 6.2.

5.6 Software stack

PIM-based accelerators [131, 13, 2, 130] are most efficient for applications that can offload a large dataset to the accelerator once and process any incoming input using the data stored in the accelerator. For example, database tables, as well as matrices for deep learning, graph, and classic machine learning applications, can be offloaded to the accelerator once and used for processing many inputs.

In all these domains, the one-time cost of pre-processing and data placement has typically been considered acceptable. For example, in graph processing, several studies [147, 149, 146, 150, 151] propose pre-processing

techniques that improve the execution time. In machine learning applications, the pre-processing time is even more negligible compared to the training cost.

Pre-processing: In Gearbox, we need to partition long columns and replicate the column offset for each partition. To balance the load, we randomize the order of columns assigned to a bank and then reorder the matrix so that the long columns and long rows are the first columns and rows of the matrix.

Data placement: For placing data, we use the offload paradigm. Therefore, an API similar to CUDA's API (cudaMemcpy()) manages the data transfer. We allocate contiguous memory space for each array in each subarray independently and then store the row address of each array as metadata. Then, in each step, we load these metadata in the Start and End latches (as shown in Figure 6.2 (c) and (d)).

Programming model: Similar to Samsung's PIM [152, 131], we are relying on a library-based programming model, where a compiler would link the kernels in computation graphs of a high-level framework (such as TensorFlow). We will release our assembly library for the evaluated kernels.

Scaling the proposed method for larger datasets:

We evaluated our approach using datasets that are as large as datasets evaluated by prior works [153, 149, 146], as evaluation with significantly larger datasets, in the slow simulation environments, is impractical. Gearbox provides high parallelism in one stack. Therefore, unlike prior works[153, 149, 146], Gearbox does not need multiple stacks for these dataset sizes.

However, to employ Gearbox for large-scale graphs, we can use multiple stacks accompanied with well-studied partitioning techniques [150, 154, 155, 156, 157]. These techniques partition the graph data into shards (i.e., coarse-grain partitions) with limited inter-shard communications. Prior work in the area of distributed graph processing software frameworks [150, 154], multi-stack PIM-based approaches [153, 149, 146], and low-capacity analog-based accelerators [132] have employed, developed, and optimized these partitioning techniques. We evaluated our approach using datasets that are as large as datasets evaluated by prior works [153, 149, 146]. Evaluation with significantly larger datasets in the slow simulation environments is impractical. To extend the architecture for larger datasets, we use multiple stacks (4-16) per device, similar to prior works [153, 149, 146]. To extend the capacity even more, we assume multiple devices will be connected by NVLink3 and NVswitch [158] or similar inter-device interconnection, which allows all-to-all bandwidth of 600 GB/s for up to 12 devices. To extend the algorithm for multiple devices and multiple stacks, we can partition the matrix into several blocks, where each block is assigned to one stack. This technique is used by prior accelerators with limited capacity [159, 132]. In this case, we require an additional step that reduces the results of all blocks. NVLink supports collective operations [160] (e.g., broadcast and allReduce operations) that efficiently support the required inter-device communications for our proposed method. We leave evaluations for multi-device Gearbox as a future work. **Supporting kernels with more than three**

arrays or more than eight instructions: SpMSpV is an example of a kernel that requires more than three arrays. Since we have only three Walkers, we break the first step of this algorithm into two steps, where each step has three arrays. Given that in-memory-layer PIM-based accelerators with high parallelism target memory-intensive application, with few instructions per loaded data, a few-entry instructions buffer is enough. For example, for our evaluated regular applications 4-entries and for SpMSpV 8-entry instruction buffer is enough. If a future work identifies a widely-used memory intensive application that require more instruction buffer entries, the instruction buffer can be extended at the cost of higher area overhead. A software solution for mapping a kernel with more than 8 instructions is to break the algorithm into few steps, similar to what we do for SpMSpV.

Handling corner cases: If the amount of remote accumulations is high, the Dispatcher SPU in the LocalAccumulations step or a Compute SPU in the Dispatching step may not find enough space for storing the received index-value pairs. To address this issue, we add a software-hardware-based mechanism. Section 5.4 explains that each Walker has an *End* latch that indicates the end of its corresponding array. When a Walker reaches the row address that is one less than the row address of the *End* latch, the SPU raises a signal that lets the logic layer know that the reserved space is about to be full. Then the logic layer controller stalls the senders (depending on the step, could be the Compute SPUs or the Dispatchers) and initiates the next step, making the array empty again.

Table 5.1: Configuration details for evaluated architectures

Component	Parameters
GPU	Tesla P100 [121], 12 GB memory 3 HBM2 memory stacks at 549 GB/s (183 GB/s per stack)
Ideal in-logic-layer GPU	512 GB/s per stack [153]
Gearbox	technology:22 nm, 32 vaults 32 subarray, open-bitline structure, 256 bytes per row, 64 banks per layer 8 memory layers, row cycle:50 ns, frequency:164 MHz in-logic-layer components per vault: 4-32 kB SRAM, an ARM Cortex-A35 [84] interconnection: 1.2 GHZ, 64 lane, latency: 0.8 ns for each interconnection segment [146, 19]

5.7 Evaluation

5.7.1 Methodology

Following prior works [132, 32, 146, 149, 153, 159], we evaluate Gearbox using three graph algorithms and two sparse machine learning kernels: Breadth-First Search (BFS), Page Rank (PR), Single-Source Shortest Path

Table 5.2: Evaluated datasets

Matrix	Full name	Rows	Non-Zeros	Density	Size (Bytes)
Holly	hollywood_2009	1139905	112751422	0.0086%	911,130,616
Orkut	soc_orkut	2997166	212698418	0.0023%	1,725,564,672
Patent	cit_Patents	3774768	33037896	0.00023%	294,501,312
Road	road_usa	23947347	57708624	0.00001%	653,247,768
Twitter	soc_twitter-2010	21297772	530051618	0.0001%	4,410,795,120

(SSSP), Sparse K-Nearest neighbors (SPKNN), and Support Vector Machine (SVM). We vary datasets to capture different characteristics of applications for different inputs. Table 5.2 introduces the datasets, which are real-world matrices from the SuiteSparse matrix collection [161], and Table 5.1 lists the configurations of the evaluated systems.

There is no established simulator for bank-level and subarray-level computing with simplified processing elements, as these approaches are only recently getting popular. Therefore, we developed an in-house event-accurate simulator for Gearbox and prior works. Furthermore, we integrated our simulator with Gunrock [162] to validate the algorithms. We further evaluate our simulator with assertion testing and analytical evaluations.

We developed the RTL model of our SPUs in 14 nm technology and incorporated an overall penalty of 3.08× for processing in 22 nm DRAM. The penalty incorporates the effect of larger technology node and other inefficiencies [98]. Consequently, we evaluated Gearbox with a frequency of 164 MHZ.

We evaluated latency, energy consumption, and area of memory elements and interconnect elements using CACTI-3DD [101]. For the breakdown of energy consumption of GPUs, we used Moveprof [119], which is a tool based on integrating NVIDIA's NVProf [108] and GPUWattch [120].

5.7.2 Speedup

Figure 5.11 compares our proposed method (GearboxV3) against a server-class GPU and a prior work, SpaceA [146]. Gearbox, with just *one* memory stack, delivers on average (up to) 15.73× (52×) speedup over a server-class GPU, NVIDIA P100, with *three* stacks of HBM2 memory. Gearbox also outperforms an ideal model of SpaceA [146], a PIM-based SpMV accelerator that only supports row-oriented processing. However, SpaceA [146] reports only 4.86% area overhead. Therefore, a fairer comparison is speedup per area. Generously assuming no area overhead, perfect load balancing, and no penalty for remote reads for SpaceA, Gearbox outperforms SpaceA, on average (up to), by 58× (447×) per area. The speedup over SpaceA stems from the fact that Gearbox provides higher parallelism and efficient support for column-oriented processing.

Although we chose Fulcrum as the baseline architecture, the two key ideas can enable column-oriented processing for all PIM approaches, such as SpaceA [146], Newton [135], and Samsung PIM [131]. Our ideas enable column-oriented processing for all PIM approaches and can improve them. For example, our ideas can speed up SpaceA by 3.4 times. To this end, we can add our hardware support for our Hybrid partitioning by adding our latches and comparators to spaceA. Similarly, we can add accumulation dispatching to SpaceA and use the bank-level CAM in SpaceA for accumulating remote results.

The speedup of Gearbox against GPU stems from three sources: (i) higher internal bandwidth compared to GPU, (ii) lower overhead for random accesses where only a few words out of a cache line is useful, and (iii) inefficiency of SIMD units in GPU for irregular applications.

Table 5.3: Each Gearbox version shown in Figure 5.12.

	Description
GearboxV0	row-oriented processing+local random access for accessing a row+ broadcasting the frontier+ using sequential index matching for processing each row
GearboxV1	column-oriented processing+column-oriented partitioning+ our proposed Accumulation dispatching
HypoGearboxV2	column-oriented processing+our Accumulation dispatching+ an impractical partitioning (partitioning the matrix with Hybrid partitioning but placing the entire input and output array in the logic layer)
GearboxV2	column-oriented processing+Accumulation dispatching+ Hybrid partitioning without replication long activating entries in each subarray
GearboxV3	column-oriented processing+reduction dispatching+ Hybrid partitioning+replicating long activating entries

5.7.3 The effect of each optimization

Figure 5.12 illustrates the effect of the proposed optimizations in Gearbox. Table 5.3 lists the description of each version. Figure 5.12 shows that GearboxV0 and GearboxV1 are three orders of magnitude and two orders of magnitude slower than Gunrock, respectively. One hypothetical version of Gearbox, HypoGearboxV2, which places the entire input and output array in the logic layer, provides, on average, 4.28× speedup compared to GPU. HypoGearboxV2 is not practical, as SRAMs in the logic layer memory elements do not have enough capacity for the entire input and output vector. GearboxV2, on average, provides 12.48× speedup over GPU by placing only long activating entries of the output/input vectors in the logic layer. The SRAM capacity for this solution is $(2 * n \times (4 + 4) * P/100$, where n is the number of rows and P is the percentage of input/out entries placed in the logic layer. For the evaluated datasets and P of 0.01 % , we need 34 KB SRAM in total in the logic layer. GearboxV3 is the final version, whose performance is discussed in Section 5.7.2.

5.7.4 Execution time and energy breakdown

Figure 5.13 (a) shows the breakdown of execution time spent on each of the six steps of the algorithm for GearboxV2 and GearboxV3. Here, most of the execution time is spent on LocalAccumulations and RemoteAccumulations. Step1 in this figure includes the overhead of broadcasting of non-zero entries placed in the logic layer, which is on average 1.1% of the total execution time.

Figure 5.13 (b) presents the breakdown of the energy consumption of Gearbox, demonstrating that Gearbox reduces the energy consumption, compared to GPU, on average (up to) by 97 (99)%. This figure shows that in most applications, row activations are the major source of energy consumption. The exception is SPKNN, where the input vector and the output vector have many non-zero values corresponding to the long columns/rows, increasing the energy consumption of the operations in the logic layer.

5.7.5 Comparison against non-in-memory-layer approaches

Figure 5.14 compares the speedup of Gearbox against three ideal models. The ideal models only account for the overhead of data movement and provide an upper bound for non-in-memory-layer approaches. Figure 5.14 shows that Gearbox provides 7.94× (31×), on average (up to), speedup per memory stack, compared to the ideal model of a GPU. We also evaluated Gearbox against a purely in-logic-layer approach under aggressive assumptions such as (i) 512 GB/s raw bandwidth, (ii) having enough parallelism to utilize the raw bandwidth, and (iii) having 56 64 kB L1 and 4 MB L2 cache to capture any locality.

Gearbox offers, on average (up to), 2.83× (11×) speedup per memory stack, compared to this ideal model of an in-logic-layer GPU. The main bottleneck of in-logic-layer approaches is the limited bandwidth in the logic layer, which is 29× lower than the bandwidth of in-memory layers. Table 5.4 compares Gearbox against a few non-in-memory layer approaches based on the reported speedup in their paper on the two common algorithms evaluated by all these accelerators (Page Rank and SSSP). The comparison overestimates the speedup of these accelerators, as we convert their reported CPU speedups to GPU speedups based on the GPU speedups reported in Graphicionado[159], which has no HBM2 memory and has half the memory bandwidth.

Tesseract [153] and GraphP [149] in Table 5.4 are using HMC-like configuration. Our speedup against these approaches shows that our speedup comes from our in-memory-layer design and not from using HMC-like configuration. Our speedup against these approaches also proves that Gearbox can outperform GPUs with Fine-Grained DRAM [163], with narrow, dedicated TSVs to each bank, similar to HMC.

Table 5.4: Speedup against non-in-memory-layer approaches.

	Graphicionado[159]	Tesseract[153]	GraphP[149]
Per stack/chip	10.01	27.08	21.99
Per area	–	13.47	10.9

5.7.6 The effect of load balancing

Figure 5.15 (a) shows that for most datasets and algorithms, labeling 0.01% of rows/columns as long can significantly improve performance. This figure also shows that increasing the percentage only slightly improves the performance.

We also evaluated the effect of distributing consecutive columns (Figure 5.15) (b). In real-world matrices, consecutive columns (e.g., neighboring nodes in a graph) are most likely to get activated together. Our evaluations show that distributing consecutive columns among subarrays in a bank (SameBank) provides, on average (up to), 22.3× (76.9×) speedup compared to storing consecutive columns in the same subarray (SameSubarray).

5.7.7 Power and temperature constraints

Figure 5.16 (a) shows that Gearbox reduces power consumption by 75%, compared to the GPU. It consumes, on average, 32.72 watts. Our power density is 465 mW/mm2, which reduces the power density of SpaceA by 12% and is safely under the power density budget of a PIM-based accelerator with a commodity-server active heat sink [164, 94] and under the power budget of the PCIe/CXL peripheral interface. We evaluated the performance of Gearbox under two power budgets: (i) 10W and (ii) 40W. Figure 5.16 (b) presents the speedup of Gearbox under these two power budgets. To lower the power consumption, we lower the frequency. This figure shows that even under a restricted power budget of 10 watts, Gearbox (with one memory stack) outperforms a high-performance GPU (with three memory stacks), on average (up to) by 6.8× (38.65×).

Table 5.5: Area evaluation of Gearbox

Component	Area mm^2			
	Per two subarrays		Per layer	
	Optimistic	Pessimistic	Optimistic	Pessimistic
Original DRAM	–	–	–	34.95
Walkers	–	0.011	–	11.26
Bank-level logic and interconnection	–	–	–	4.56
Integer SPUs	0.0067	0.010	6.86	10.42
Float SPUs	0.0098	0.019	10.03	19.45

5.7.8 Area evaluation

Table 5.5 lists the optimistic and pessimistic areas of our hardware components. Our optimistic area numbers are the ones reported by our synthesizer, scaled to 22nm. Our pessimistic area evaluation is the maximum of scaling the optimistic area for 4 layers (using the scale factor derived from [165]) and the pessimistic area reported by our synthesizer. For Walkers, we evaluate the area using CACTI-3DD [166], which is equivalent to pessimistic area evaluations. Gearbox optimistically (pessimistically) imposes 2.42% (10.93)% area overhead compared to a prior work, Fulcrum [13]. In comparison with regular HMC memory, Gearbox optimistically (pessimistically) imposes 73% (100)% area overhead.

5.7.9 Evaluation for regular kernels

GearBox is based on Fulcrum. Therefore, GearBox/Fulcrum can also support and speed up regular workloads. Figure 5.17 evaluates performance for a range of regular applications from the InSituBench [13] suit. For these evaluations, both Gearbox/Fulcrum and our bank-level SIMD have the same number of ALUs and have the same frequency. Gearbox provides, on average, 4.4× higher throughput than the bank-level SIMD approach. Gearbox can also outperform DRISA [2], a row-wide bitwise-based SIMD approach, which implements arithmetic operations using bit-wise operations on horizontally laid-out data, by more than two orders of magnitude. SIMDRAM [130], another row-wide bitwise-based SIMD approach that implements arithmetic orations on vertically laid out data, cannot support floating-point operations of the evaluated applications. The vertical layout is also highly inefficient for random accesses, as we would have to activate 32 rows to access a single 32-bit word, one bit per row (the rest of bits in all rows will not be used).

5.7.10 Evaluation per dataset

We varied datasets to capture different characteristics of applications for different inputs. Figure 5.18 illustrates the speedup achieved for each algorithm and each dataset listed in 5.4.

5.8 Related Work

SIMD and row-wide bitwise approaches: Bank-level SIMD approaches [131, 135] or subarray-level bit-parallel and bit-serial approaches [130, 2, 12] perform the same operation on multiple aligned words. These approaches cannot efficiently support SpMV and SpMSpV. Section 5.7.9 compares Gearbox against these approaches for regular kernels.

Logic-layer-based approaches: This category of prior works [153, 149, 167, 168] employs a few processing units with traditional or decoupled access/execute architectures[74] in the logic layer. These approaches still move data along subarrays, banks, and layers, imposing data movement overheads. Section 5.7.5 discusses these approaches.

NVM-based techniques: These approaches employ NVM computation capabilities (e.g., CAM capability, analog MAC, and digital computation capabilities) [32, 132, 169]. Due to several issues with NVM-based approaches, including the hardware and energy overhead of analog-to-digital/digital-to-analog converters (which can limit the capacity to 64 MB [132]), low endurance, and high error rate, in this paper, we have focused on DRAM-based accelerators.

ASIC/FPGA-based accelerators: Several ASIC and FPGA designs [133, 170, 159, 171, 134] target SpMSpV and graph processing. The advantage of these approaches is that their performance, similar to other non-PIM approaches, does not highly depend on the data placement in memory. Therefore, they do not require careful offline data placement. For example, they can handle load imbalance at runtime by distributing tasks among processing elements. However, these approaches have to transfer data from memory to the accelerator, imposing data movement overhead. We evaluate Gearbox against Graphicionado [159], an ASIC-based approach, in Table 5.4.

5.9 Conclusions

Gearbox extends the range of applications that highly parallel PIM-based accelerators can support, by proposing hardware support for Accumulation dispatching, Hybrid partitioning, and subarray-level random accesses.

We can envision three types of future works: (i) extending Gearbox for other irregular kernels, (ii) applying Gearbox in an SRAM/EDRAM setting, (iii) augmenting Gearbox with a reliability mechanisms for memory technologies with higher error rate. (In this work, we employ Gearbox for graph processing, which is tolerant to error and uses DRAM, where the probability of error per byte in one month, in memory layers, is as low as 1.86375e-8 (85% of DRAM errors caused by the memory controller and memory channel [172])).

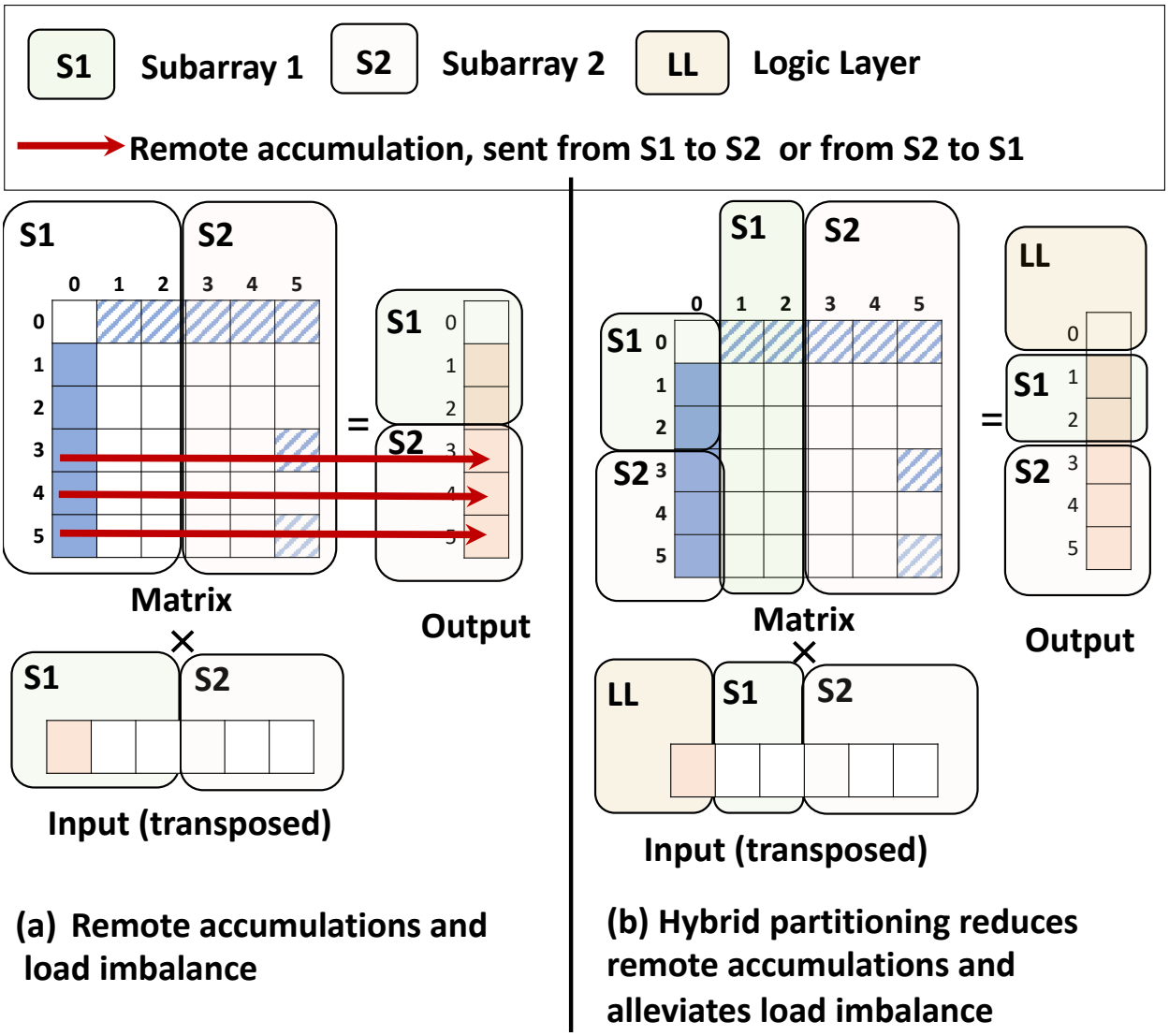


Figure 5.2: Remote accumulations and load imbalance. (a) With column-oriented partitioning, long columns cause load imbalance and many remote accumulations. (b) With Hybrid Partitioning, long column entries cause no remote accumulation and no load imbalance.

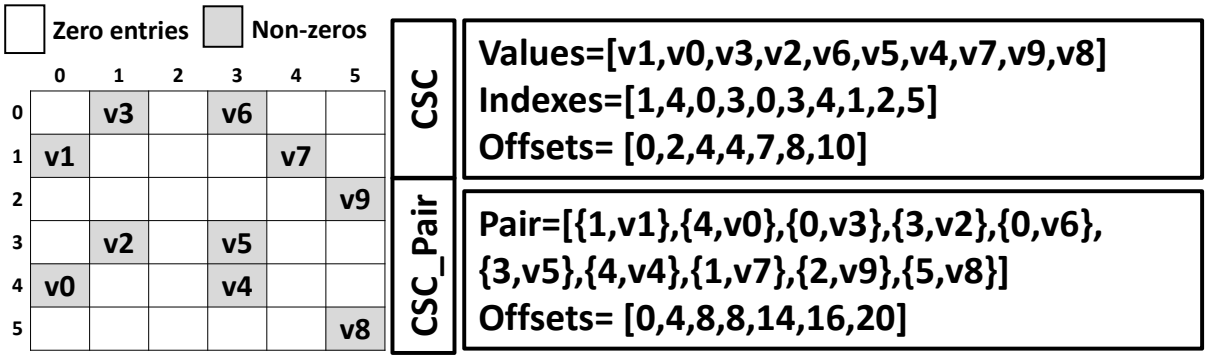
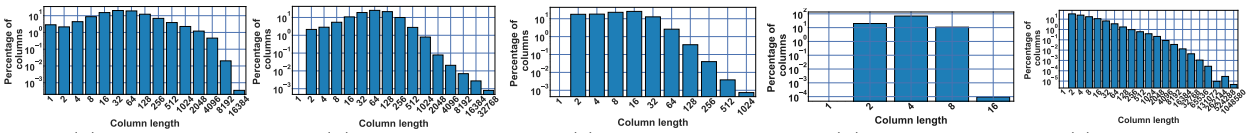


Figure 5.3: A sparse matrix in CSC and CSC_Pair format.



(a) Holly (b) Orkut (c) Patent (d) Road (e) Twitter

Figure 5.4: Column length distribution in real-world matrices (both x and y-axis are in log scale).

```
Input:
    CSC_offsets[0:n]
    CSC_Pair[0: numNonZeros(Matrix)*2-1],
    //pair sparse format of the input vector
    currFrontier[0: numNonZeros(Input)*2-1]
Output:
    OutputDense[0:n-1]=0
    numOutputNonZeros=0
    //pair sparse format of the output vector
    nxtFrontier[0: 2*n-1]
//processing only columns corresopnding to non-zeros of the input
for (i=0; i< numNonZeros(Input)*2; i+=2):
    f_column= currFrontier[i]
    f_value= currFrontier[i+1]
    col_offset= CSC_offsets[f_column]
    col_length= CSC_offsets[f_column+1]-CSC_offsets[f_column]
    for (j=0; j< col_length; j+=2):
        row_index= CSC_Pair[col_offset +j];
        row_value= CSC_Pair[col_offset+j+1]
        //random write to the output vector
        OutputDense[row_index]+= row_value * f_value
//generating the sparse format of the output vector
for (i=0; i<n, i++):
    If(OutputDense[i]!=0):
        t= numOutputNonZeros*2
        nxtFrontier[t]= i
        nxtFrontier[t+1]= OutputDense[i]
        numOutputNonZeros++
```

Figure 5.5: The column-oriented algorithm.

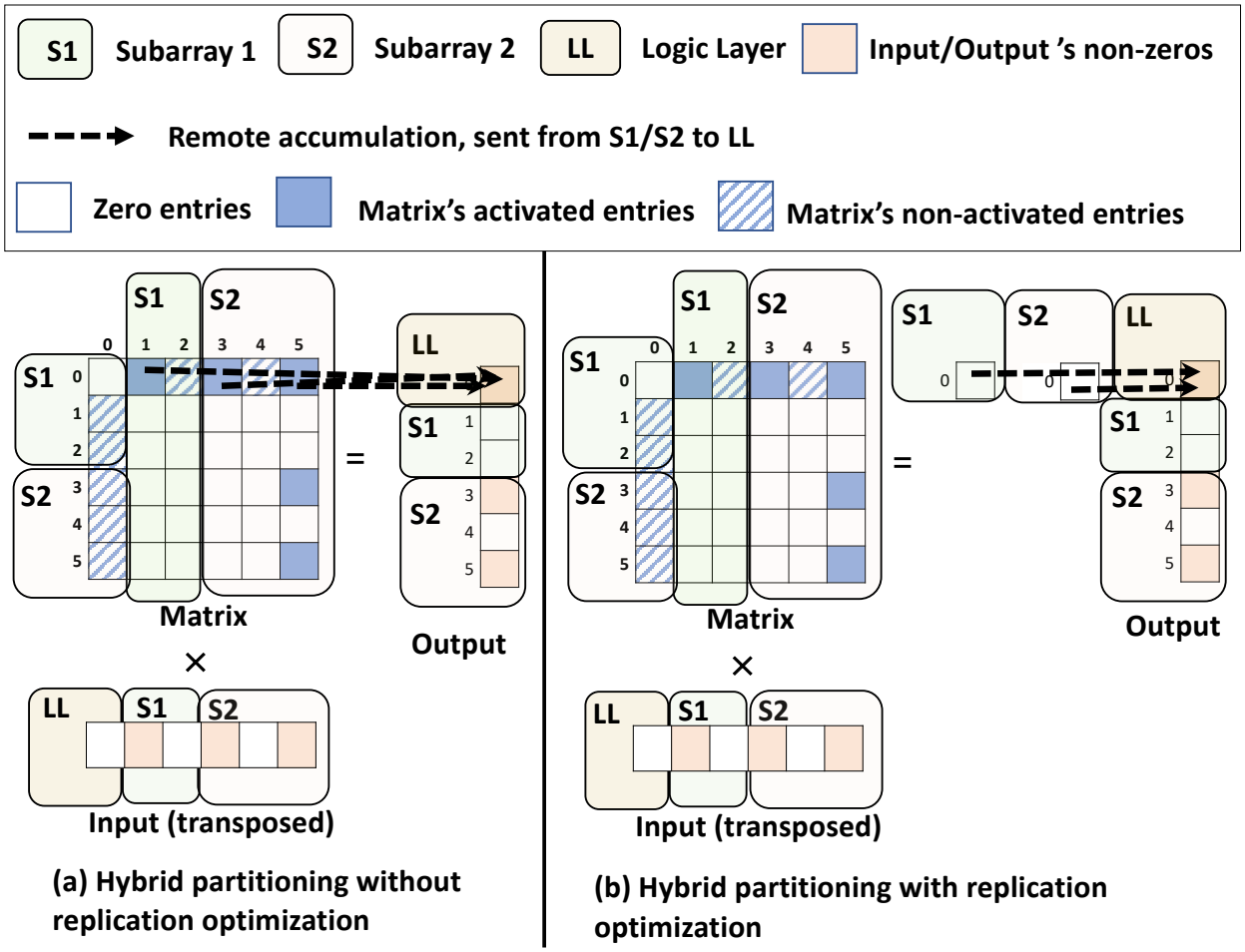


Figure 5.6: An extra optimization that replicates the output vector entries corresponding to long rows/columns in each subarray.

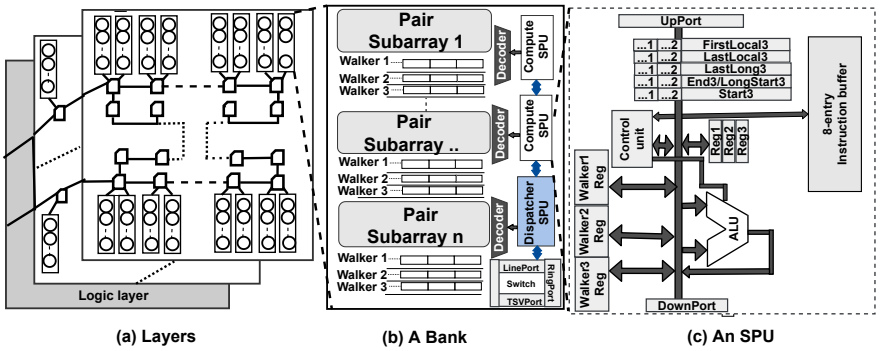


Figure 5.7: Overall architecture. In (a), the circles are subarrays, the rectangles are banks, and the pentagons are switches.

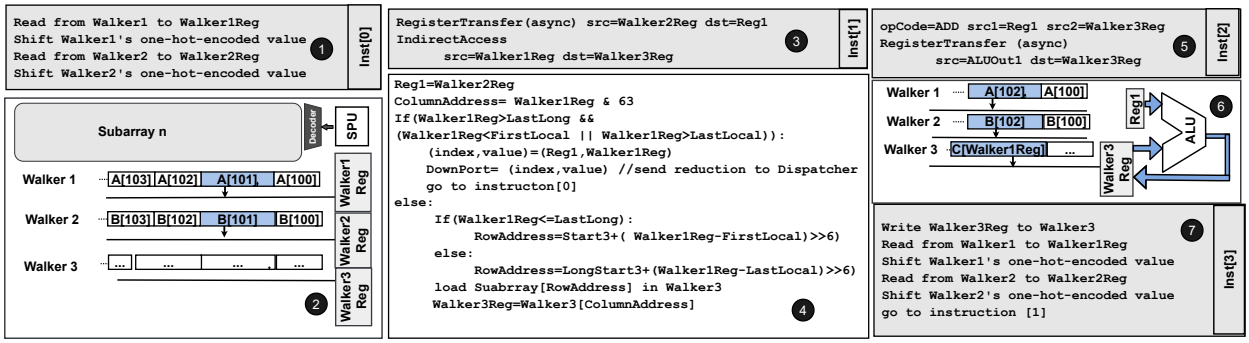


Figure 5.8: A walk-through example for $C[A[i]] += B[i]$ with four instructions.

```
1   Input:
2       CSC_offsets[0:n]
3       //pair sparse format of the input vector
4       frontier[0: numNonZeros(Input)*2-1]
5   Output:
6         pack[0: packLength-1]
7         packLength
8   j=0;
9   // pack frontier value with correponding column information
10  for (i=0; i< numNonZeros(Input)*2; i=i+2):
11      index= frontier[i]
12      pack[j]= CSC_offsets[index]
13      pack[j+1]= CSC_offsets[index+1]-CSC_offsets[index]
14      pack[j+2]= frontier[i+1]
15      j=j+3
16  packLength=j+1
```

Figure 5.9: OffsetPacking.

```
1   Input:
2         pack[0 : numNonZeros(Input)*3-1]
3         CSC_Pair[0: numNonZeros(Matrix)*2-1]
4   Output:
5         OutputDense[0:n-1]
6   OutputDense[:]=0
7   for (i=0; i< numNonZeros(Input)*3; i=i+3):
8       offset=pack[i]
9       length= pack[i+1]
10      f_Value= pack[i+2]
11      for (j=0; j< length;j=j+2):
12          row_index= CSC_Pair[offset+j]
13          row_value= CSC_Pair[offset+j+1]
14          //the fisrt step for generating nxtFrontier
15          if(OutputDense[row_index]==0)
16              send (0,row_index) to the dispatcher
17          OutputDense[row_index] +=f_Value* row_value
```

Figure 5.10: LocalAccumulations.

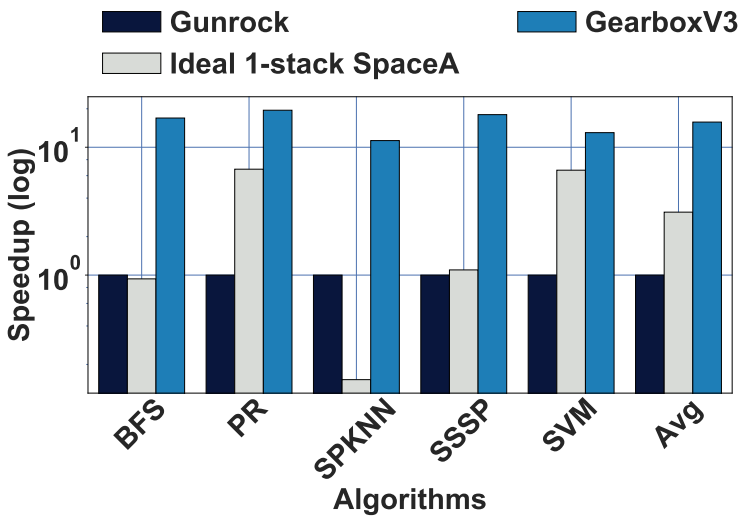


Figure 5.11: Average speedup of our final solution (GearboxV3) against a GPU framework (Gunrock) and a prior work (SpaceA). The values less than 10^0 represent slowdown.

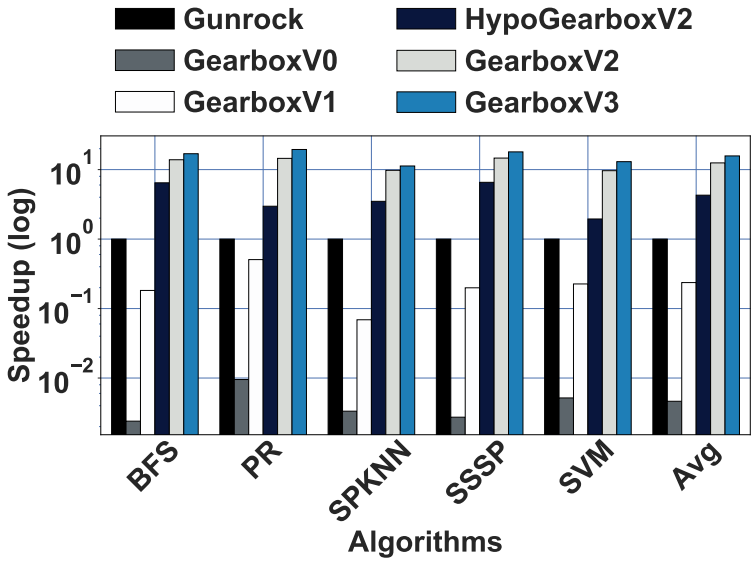
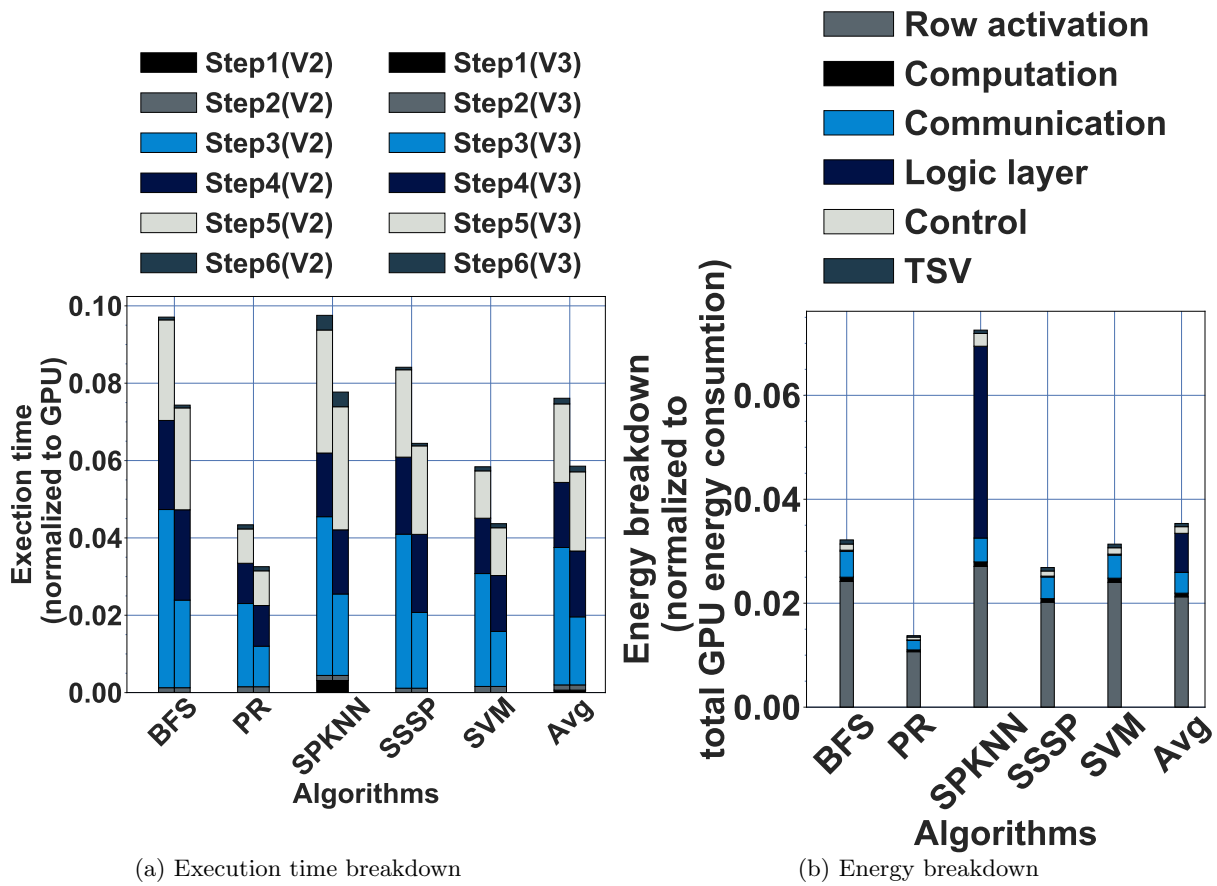


Figure 5.12: The effect of each optimization. Table 5.3 lists the description of each Gearbox version.



(a) Execution time breakdown

(b) Energy breakdown

Figure 5.13: Breakdown of execution time and energy

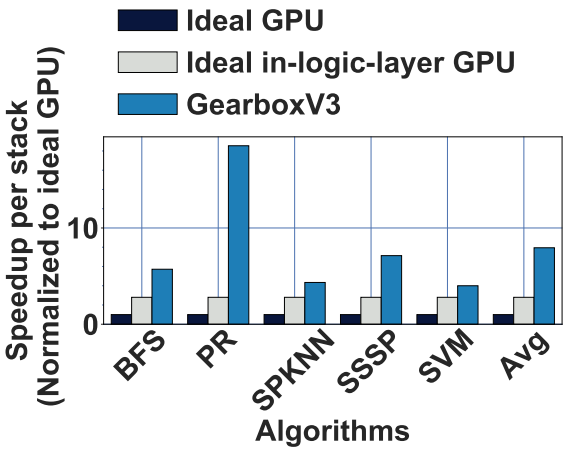
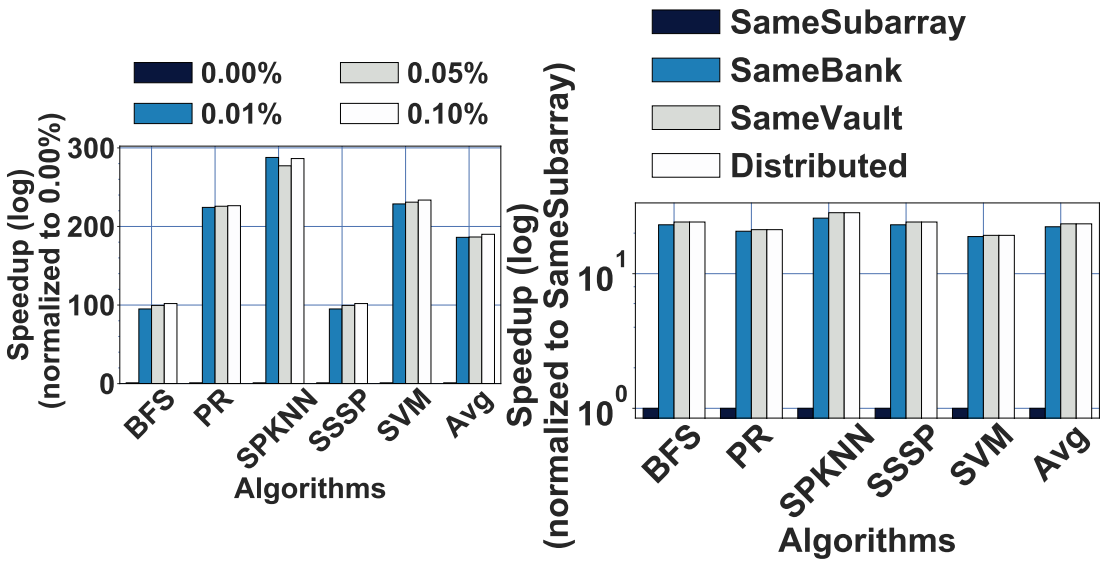


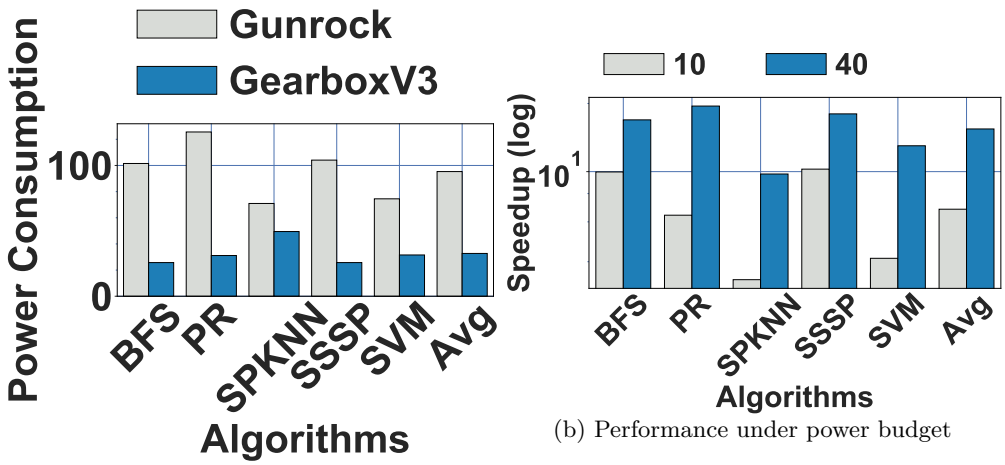
Figure 5.14: Comparison against ideal models.



(a) Percentage of long rows/columns

(b) Consecutive columns

Figure 5.15: (a) The effect of load balancing techniques.



(a) Power consumption

(b) Performance under power budget

Figure 5.16: Power and temperature constraints.

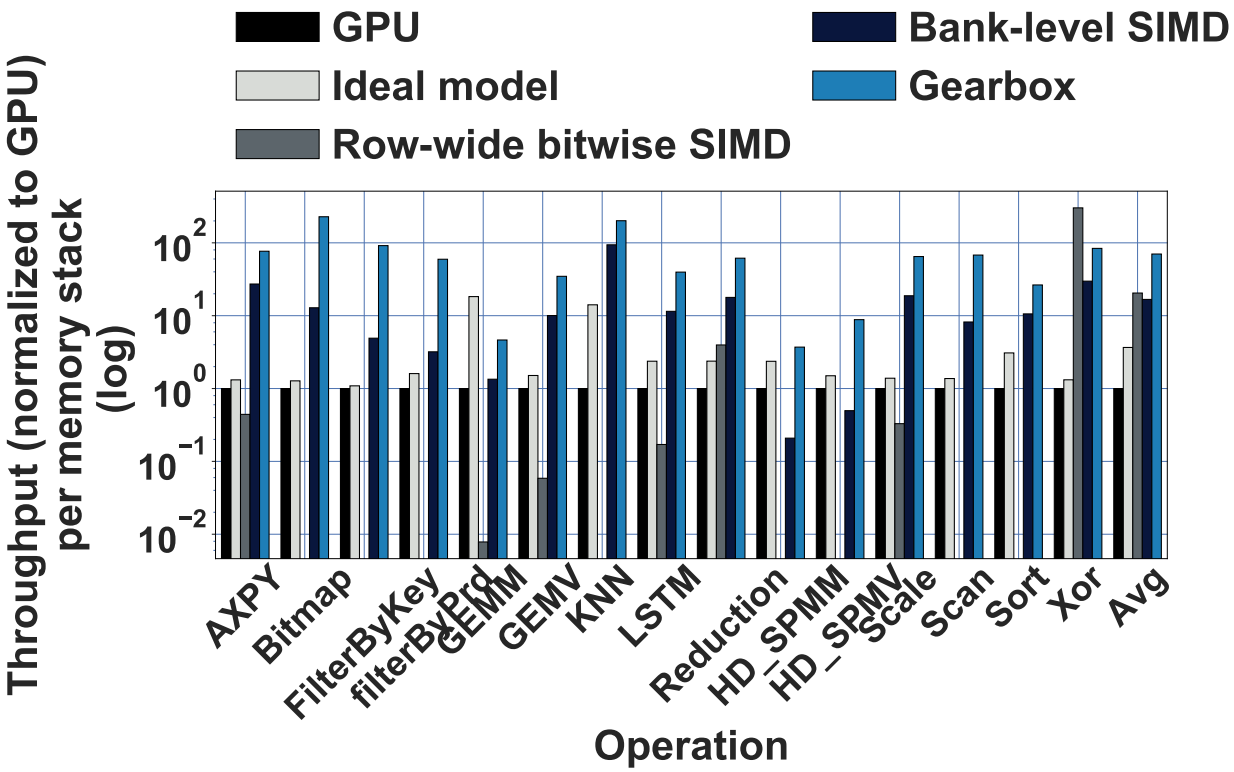
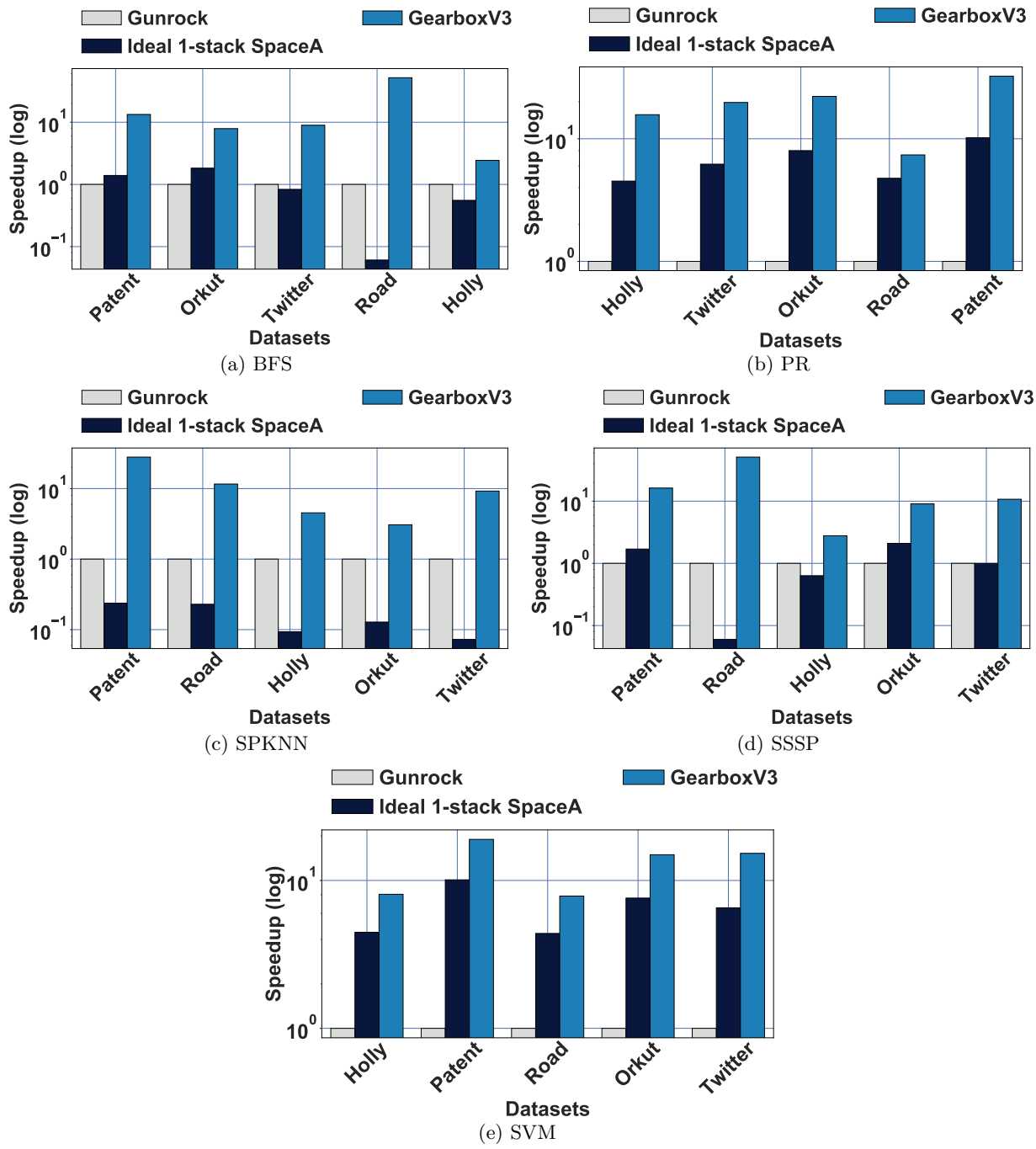


Figure 5.17: Speedup for regular kernels.



(a) BFS (b) PR

(c) SPKNN (d) SSSP

(e) SVM

Figure 5.18: The speedup of our final solution (GearboxV3) against a GPU framework (Gunrock) and a prior work (SpaceA) for each dataset and algorithm.

Chapter 6

Pulley: An Algorithm/Hardware Co-optimization for Multi-device In-memory Sorting

6.1 Introduction

In current computing systems, the latency and energy consumption of fetching data from off-chip memory can be 2-3 orders of magnitude higher than an arithmetic operation [4]. Processing-in-memory (PIM) architectures alleviate this data movement overhead by processing data inside the memory.

Sorting is an important task that requires many passes over data, where each pass performs only a few operations per loaded datum from memory. As a result, for datasets too large for the processor's caches, the overhead of moving data from memory to processor is much higher than the overhead of computation for sorting algorithms.

Sorting is widely used and appears in many big data applications and database operations, such as index creation, sort-merge joins, and user-requested output sorting [173]. Accordingly, many studies have focused on accelerating sorting using GPU, FPGA, and ASIC [173, 16, 17, 174]. Sorting is thus a natural candidate for PIM acceleration.

Among sorting algorithms, radix sorting is highly parallel and can exploit the parallelism of PIM. With the emergence of high-bandwidth interconnect (e.g., NVLink, and CXL) (which enable even more parallel multi-device and multi-stack PIM-based accelerators), it is crucial to employ highly parallel and scalable algorithms such as radix sorting.

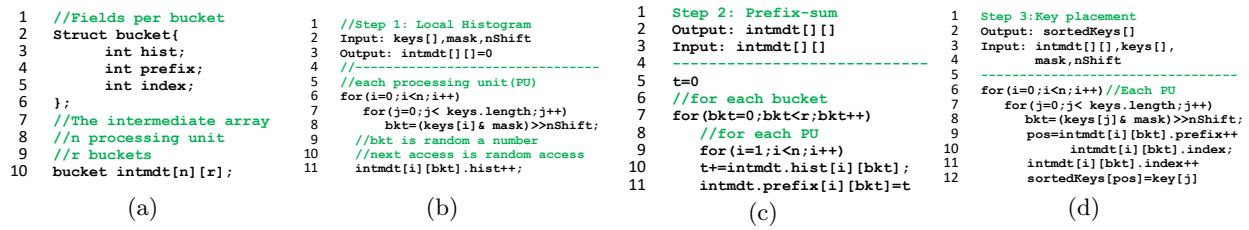


Figure 6.1: Parallel Radix sorting: (a) the structure of the intermediate array, (b) in step 1 each processing unit (PU) generates a local histogram array, (c) in step 2, we need a prefix-sum on all local histogram arrays, and (d) in step 3, each PU determines the location of each key by deriving the bucket number (line 8) and adding the prefix value of the bucket to the current index of the bucket (line 9).

Radix sorting comprises several passes. In each pass, each processing unit places a key in a bucket in three steps. First, the Histogram generation step, where each processing unit generates a histogram array by counting the number of keys in each bucket. Second, the Prefix-sum step, where the algorithm requires a prefix sum across all local histogram arrays generated by all processing elements. Finally, the Key placement step, where each processing uses the prefix-sum array for finding the address of each key in the sorted sequence and writing it in the correct address.

Implementing Radix sorting in PIM is challenging for three reasons. First, the histogram generation step introduces random accesses and is most efficient with fine-grained random access memories (e.g., SRAMs). PIM units often lack such memory elements. In fact, memory reads the data at a row granularity, which is often quite large, several Kbits. As a result, random accesses to the memory segments are very costly because each random access reads an entire row of data. Second, the larger the number of buckets in each pass, the fewer the numbers of passes (more details in Section 6.2.1). Accordingly, to limit the number of passes, we need to reserve a large histogram array per processing element, wasting the capacity. Third, radix sorting requires prefix-sum across all memory segments, which impose significant overhead if performed using a core far from memory segments.

In this paper, we propose an algorithm and hardware co-optimization for sorting, Pulley, that (i) reduces the number of required intermediate arrays and (ii) eliminates random accesses.

To reduce the number of required intermediate arrays, we propose hardware and software optimization, by which every group of processing units can efficiently share an intermediate array. Since we share the intermediate array among every m processing element, the number of intermediates arrays decreases, decreasing the capacity overhead of intermediate arrays and the performance overhead of the prefix-sum operations.

To eliminate the random accesses, we employ a baseline PIM architecture, Fulcrum [13], that is highly efficient for the operation of dichotomizing keys into two groups (more details in Section 6.3.1). We optimize

this baseline architecture for locally sorting keys in each memory segment using binary radix sorting. Our modified algorithm exploits locally sorted keys and performs the histogram generation step only with sequential accesses.

Our paper makes the following contributions:

- Proposing an algorithm/hardware co-optimization approach for enabling large-scale sorting on PIM-based accelerators.
- Evaluating the effect of our proposed method against a near-HBM FPGA-based approach [16] and an in-logic-layer-based sorting accelerator [17].

6.2 Background and Motivation

6.2.1 Parallel Radix Sorting

Radix sorting is a sorting algorithm that is highly parallel and can exploit the parallelism of recently proposed PIM-based accelerators, where each memory segment has one processing element.

This algorithm splits the k bits of keys into smaller d-bit digits and sorts data in $\lceil k/d \rceil$ passes. In each pass, the algorithm partitions the keys into $radix = 2^d$ distinct buckets. Therefore, the fewer passes, the larger the number of buckets. For example, for 32-bit keys, if we uses $radix = 2^{16}$, we need only two passes but2^{16} buckets. As explained in the introduction, radix sorting requires 2^{16} buckets. Figure 6.1 (a) shows the structure of the intermediate array, which has one element per bucket for each processing unit. To save the memory space, the prefix-sum can be performed in place so that we only need one field for histogram and prefix-sum values. However, in the Key placement step, we need another intermediate value (field) that keeps the current index of the bucket. Accordingly, we need to reserve at least 542 MB per processing element for the intermediate arrays.

In addition to the capacity overheads, the intermediate arrays impose performance overhead due to (i) random accesses and (ii) the prefix-sum operation. Figure 6.1 (b), (c), and (d) illustrates how the sorting algorithm uses the intermediate array in each step. Line 11 in Figure 6.1 (b) and line 12 in 6.1 (d) also shows that radix sorting requires random access to the intermediate array. Line 7 and 9 in Figure 6.1 (c) show that with inefficient implementation, the overhead of the prefix-sum operations can be in the order of $n \times r$, where n is the number of processing elements, and r is the number of buckets. In this paper, we reduce the overhead of random accesses and the overhead of prefix-sum operations.

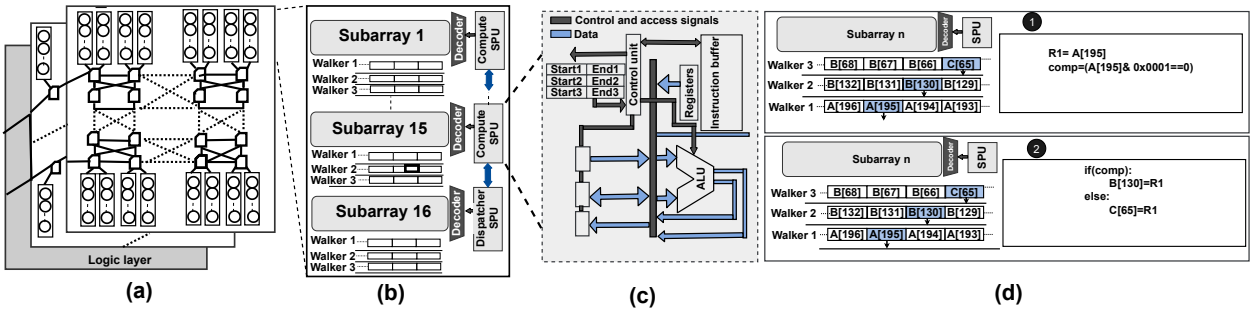


Figure 6.2: Our proposed architecture: (a) The circles are subarrays, the rectangles are banks, and the pentagons are switches. Banks and subarray connected using a interconnection with dragonfly topology (b) a bank with an SPU and three Walkers per subarray, (c) architecture of each SPU, and (d) an example of local binary radix sorting.

6.3 Our proposed method

Figure 6.2 illustrates the architecture of our proposed method, where the memory stack has a few layers; within each layer, banks are connected through an interconnection network with a dragon-fly topology, and within each bank, subarrays are connected through a line interconnection topology.

6.3.1 Baseline PIM architecture

We use a subarray-level PIM approach, Fulcrum[13], as the baseline PIM architecture. In Fulcrum each pair of subarrays have one simplified sequential processing unit (Figure 6.2 (b)). Each subarray-level processing unit (SPU) has a few registers, an 8-entry instruction buffer, a controller, and an ALU (Figure 6.2 (c)). The simplified design is motivated by the characteristics of memory-intensive applications, where there are few simple operations per loaded datum.

In Fulcrum, every subarray pair also has three row-wide buffers, called Walkers. The Walkers load an entire row from the subarray at once, but the processing units sequentially access and process one word at a time. The sequential access is enabled by using a one-hot-encoded value, where the set bit in this value selects the accessed word. Therefore, to sequentially process the row, the processing unit only needs to shift the one-hot encoded value.

6.3.2 Fulcrum's shortcomings for Sorting

Fulcrum [13] employs its efficient local sequential processing for in-memory sorting. In this approach, first, data is bucketed among subarrays based on the most significant bits, assuming no bucket is larger than the reserved size for buckets in each subarray. This bucketing is performed by iteratively dichotomizing keys into two groups and transferring one group from one subarray to another, using an efficient inter-subarray data

movement mechanism [2]. Then, data in each subarray is sorted using binary radix sorting ($r = 2$). Again, for each bucket of the binary radix sorting, Fulcrum reserve a space and assumes that no bucket is larger than the reserved size for buckets. Accordingly, Fulcrum can only efficiently sort Megabytes of data uniformly distributed among buckets and wastes capacity for Gigabytes and non-uniformly distributed keys.

Moreover, Fulcrum cannot efficiently support sorting across banks and only sort data within a bank, assuming data is already is bucketed among banks during data transfer among the host and the accelerator. This assumption is not true in many scenarios.

In this paper, we address Fulcrum's shortcoming by enabling radix sorting that uses histogram and index arrays for finding the exact address of each key in a sorted stream, eliminating the need for reserving any extra space for each bucket.

To reduce the number of passes, we use use the radix of 2^{16}. Therefore, for 32-bit/64-bit keys we require two/four passes of bucketization on data, where each pass comprises four steps: (i) Local sorting, (ii) Histogram generation, (iii) Prefix-sum, (iv) Merging and key placement. In the following subsection, we will explain our contribution in each step.

6.3.3 Local sorting

The three Walkers with shift-based sequential access mechanisms are highly efficient for binary radix sorting, where we need to dichotomies an array of keys into two buckets (Bucket0 and Bucket2). To this end, we load the key array row-by-row in Walker1, employ Walker 2 as Bucket0 and Walker3 as bucket1. Then, as shown in Figure 6.2 (d), in each clock cycle, the SPU shifts the one-hot-encoded value to read one key from Walker1 and write it to either Walker2 or Walker3 based on the digit being processed by shifting the one-hot-encoded value of the corresponding Walker. The binary radix sorting is efficient because it requires no random access. The only problem is that, with non-uniformly distributed keys, the size of each binary bucket can be very different in each pass. To address these issues, instead of reserving a large space for each binary bucket, we propose to reserve a space that is almost the size of the key array. Then, we design a hardware controller that starts Bucket0 from the bottom of the space and fills it upward and starts Bucket1 from the end of the space and fills it downward. The reverse ordering of keys in Bucket1 can violate stability, a requirement for radix sorting. Therefore, our controller processes Bucket1 from end to start.

6.3.4 Histogram generation

With radix of 2^{16} the required space for histogram and index values is 542 MB per processing element. We propose that 15 processing units of 15 subarrays in a bank share a space for the intermediate array in the

lower subarray in the bank. This step comprises three substeps: (i) each SPU generates the histogram values of the first 256 buckets, (ii) reduce the histogram values of each of these buckets in the lower subarray, (ii) go to sub-step (i) to generate the histogram values of the next 256 buckets until the histogram values of all the 2^{16} buckets are processed.

As we explained, the second sub-step requires reducing the histogram values of 256 buckets. We can perform the reduction by a processing unit at the bank's edge that reads data from all 15 subarrays and performs the required operations. When the number of data elements per subarray is small, the overhead of this approach is negligible. However, when the number of data elements per subarray is small, the overhead becomes significant. Therefore, to propose an approach that is efficient for any number of data elements per subarray, we propose Cooperative operations, where 15 processing elements cooperate to reduce their histogram values in the last subarray of the bank.

Assuming the histogram array in i^{th} subarray is $Hist[i][:]$, the Cooperative reduction is as follows: the i^{th} processing unit receives a value from $(i-1)^{th}$ processing unit, adds it to its Hist[i][j] entry and passes the result to the $(i+1)^{th}$ processing element.

6.3.5 Prefix-sum

In a 3d-stacked memory, in our configuration, 256 subarrays share a bus (TSVs). Current PIM approaches require reading histogram values from all these subarrays through the shared bus and writing the prefix-sum values back to all 256 subarrays. Since we reduce the number of intermediate arrays, the overhead of prefix-sum decreases to reading and writing to only 16 subarrays in a vault.

Similar to reduction operations explained in Section 6.3.4, when the number of data elements per subarray is small, the overhead of this approach is negligible. However, when the number of data elements per subarray is small, the overhead becomes significant. Therefore, to propose an approach that is efficient for any number of data elements per subarray, we can use Cooperative prefix sum, where all processing elements in a memory stack cooperate to reduce their histogram values in the last subarray of the bank.

Assuming the histogram array in i^{th} memory segment is $Hist[i][:]$ and the prefix-sum array in i^{th} memory segment is $Prefix[i][:]$ the Cooperative prefix sum is as follows: the i^{th} memory segment receives the j^{th} entry of prefixe-sum ($Prefix[i-1][j]$) from $(i-1)^{th}$ memory segment. The i^{th} memory segment adds $Prefix[i-1][j]$) to its $Hist[i][j]$ to form the $Prefix[i][j]$ and forwards $Prefix[i][j]$ to $(i+1)^{th}$ memory segment.

6.3.6 Merging and key placement

Since, in our proposed method, all the 15 subarrays in a bank share an intermediate array, we need to merge the sorted keys in these arrays for correct key placement by deriving the address of each key using the prefix-sum values (as shown in Line of Figure 6.1) (d). However, naive merging imposes significant overhead because out of 16 SPUs only the last SPU will perform the merging, decreasing the parallelism. Our Cooperative merging addresses this issue. Using our proposed Cooperative merging, i^{th} SPU in a bank receives a key, compares it with its current key. If the received key is less than its current key, the SPU forwards the received key. Otherwise, the SPU forwards its current key. Once the minimum key reaches the lower SPU in the bank, this SPU derives the address of each key using the prefix-sum values.

6.4 Evaluation

6.4.1 Methodology

Pulley targets sorting Gigabytes of data, where the capacity of one memory stack is not enough. Given that new interconnection technologies, such as NVLink [158] provide high-bandwidth fully-connected topology among multiple devices, we can increase the capacity of our accelerator by connecting multiple devices. We evaluated Pulley in a 6-device setting, where each device has four stacks of 8-GB memories, providing 192 GB capacity. We chose to place four stacks per device to ensure that the power consumption of each device is less than 300 Watt. We chose the 6-device setting because the second generation of NVLink allows 6 Links per device. The third and the fourth generations allow 12 and 18 links per device. However, our current simulation environment cannot evaluate significantly larger datasets. For the same reason, we only evaluate 24 GB of data but distribute data among the 24 stacks of our 6-device configuration to incorporate the overhead of communications among devices and stacks. For each stack, we follow the configurations of fulcrum [13]. We developed an in-house event-accurate simulator for Pulley and will release the source code of the simulator.

6.4.2 Throughput

We compared our proposed method against an (i) state-of-the-art near-HBM FPGA-based sorting accelerator Bonsai [16]) and an in-logic-layer sorting accelerator (IMC-Sort [17]). The two evaluated approaches are the most related works that can support sorting Gigabytes of data.

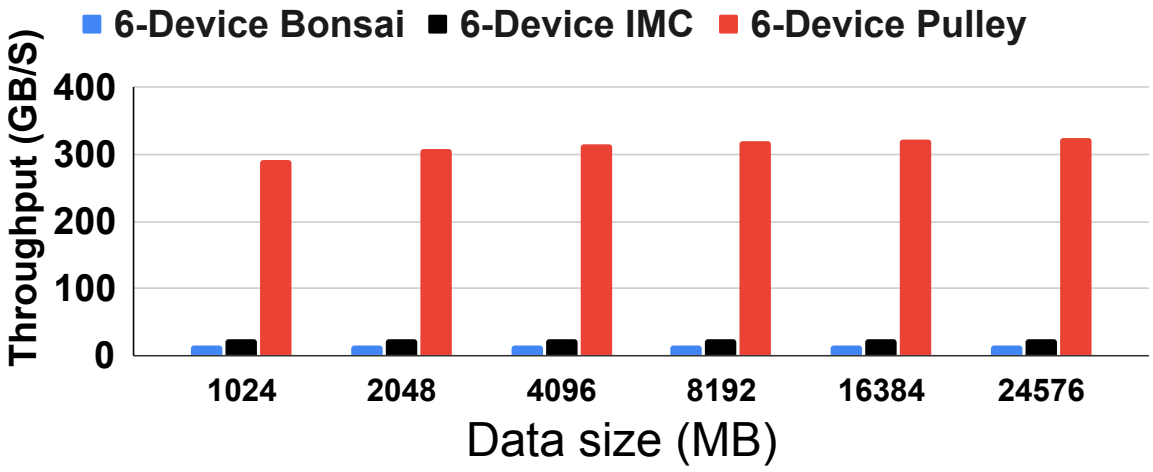


Figure 6.3: The throughput comparison.

6.4.3 Power and temperature constraints

We evaluated the energy consumption of the memory elements and interconnect elements in Pulley using CACTI3DD [166] and evaluate the energy consumption of processing units using the RTL synthesize. The average power consumption of Pulley per stack is 38.6 watts. Our average power density is 540 mW/mm2, which is under the power density budget of a PIM-based accelerator with a high-end server active cooling (1214 mW/mm2 [146, 164, 94]) and under the power budget of the PCIe peripheral interface (300 Watts per device and 75 per stack).

We could not add performance per watt against Bonsai [16] and IMC. Bonsai provides no energy number. IMC [17] only provides energy numbers normalized to CPU, and we do not know what the absolute energy consumption of this method is.

6.5 Conclusions and future Work

This paper motivates providing hardware support for sharing intermediate arrays in sorting. As future works, we envision investigating the benefits of shared intermediate arrays for other important kernels such as graph processing and database operations. We also optimized operations on the shared intermediate by providing hardware support for Cooperative operations. Future works can investigate what other applications can benefit from these operations.

Chapter 7

Conclusions and Future Works

For memory-intensive tasks, data movement dominates computation. To reduce the cost of data movement, we have two options: (i) reducing the size of data, (ii) processing data inside the memory. This dissertation explores both approaches.

This dissertation, first, presents a light-weight compression method by exploiting quantization in general-purpose processors. Our method represents values with few bits to reduce the size of data. It ensures that values that require more than a few bits are represented in their original format, preventing overflows and limiting the maximum absolute error. To this end, we propose a software-hardware interface whereby we transfer information of our high-level abstraction to hardware modules and transparently convert the short values to the original format before being used in the computation. Our evaluation demonstrates that our light-weight compression provides 1.23× speed up compared to the state-of-the-art cache compression techniques and can significantly reduce the cost of data movement, supporting the first part of our hypothesis.

Next, this dissertation explores the second approach by presenting an in-situ PIM design that keeps computations close to the subarray's row buffer to avoid substantial data-movement overheads. Our proposed method, Fulcrum, overcomes key limitations of prior in-situ architectures by placing a scalar, full-word processing unit at the edge of each pair of subarrays. We show that sequentially processing a row (instead of bit-parallel processing of the entire row buffer) with full-word computation ability allows a much wider range of tasks to leverage in-situ processing, including a full range of arithmetic operations, essential sparse and dense linear algebra tasks, operations with data dependencies, operations based on a predicate, scans and reductions, and so forth. Our evaluations show that Fulcrum outperforms a server-class GPU with three stacks of HBM2 and, on average, provides 70× speedup per memory stack and reduces the energy

consumption by 96%. Therefore, our near-data processing approach can significantly reduce the cost of data movement, supporting the second part of our hypothesis.

Continuing exploring the PIM approach, we observe that rethinking communication and load balancing mechanisms can unlock the benefits of PIM for a wider range of applications. Accordingly, this dissertation presents Gearbox, where we augment Fulcrum with hardware support for (i) offloading accumulations operations from one memory segment to another memory segment and (ii) balancing the load among processing elements. We find that Gearbox can outperform Gunrock, a GPU-based graph-processing framework, by 15.73×, showing that near-data processing can significantly reduce the cost of data movement for graph processing applications, supporting the second part of our hypothesis.

Finally, we extended Fulcrum to support multi-device in-memory sorting by adding hardware support that enables every group of processing elements cooperatively generate an intermediate array, eliminating the capacity overhead of reserving one intermediate array per processing element. Our evaluations show that Pulley (FulcrumV3), on average, delivers 20 *times* speedup compared to Bonsai, an FPGA-based sorting accelerator, and can significantly reduce the cost of data movement by processing near data, supporting the second part of our hypothesis.

Lesson Learned and Future Works

After exploring the effect of the quantized memory hierarchy in general-purpose CPUs, we realized CPUs could not efficiently exploit the higher effective memory bandwidth provided by our quantized memory hierarchy. If we had known this fact, we would have proposed a quantized memory hierarchy for SIMD or SIMT architectures that can exploit the higher memory bandwidth.

Recent works show that, for few-bit integer operations, such as 4-bit multiplications, the row-wide bitwise approaches are highly efficient. If we had it to do over again, we would have designed an architecture that could exploit the benefits of both approaches.

We envision six types of future works: (i) evaluating the effect of quantized memory hierarchy for GPGPU, SIMD and embedded processors, (ii) developing software stacks for simplified in-situ designs, (iii) mapping new processing or pre-processing to Fulcrum, (iv) evaluating and optimizing Fulcrum for other memory technologies, (v) enhancing the reliability and security of system using Fulcrum, (vi) designing a subarray-level architecture that supports both row-wide bitwise operations and single-word processing units.

- **Evaluating the effect of quantized memory hierarchy for GPGPU, SIMD and embedded processors:** Cost of data movement is much higher for GPGPU and SIMD applications as they require a higher memory bandwidth. Cost of data movement is also high for consumer devices where we use a

low-bandwidth memory. We believe with slight modifications, we can employ and evaluate an overflow-free quantized memory hierarchy in these processors.

- **Developing a software stack:** We envision a software stack composed of two steps. The first step is to implement the important kernels of the most commonly used libraries. The second step is to use automatic tools that map Data-Flow Graphs (DFG), where each operator node can have multi-dimensional vectors or tensors into a collection of primitives and kernels, which are implemented in the first step.
- **Mapping new processing and pre-processing:** We believe that many kernels can be accelerated using Fulcrum. Thus, researchers in the domain of high-performance computing can use Fulcrum to map the memory-intensive parts of their applications to Fulcrum and evaluate the end-to-end speedup. Fulcrum has also the potential to aid as a pre-processor that eliminates unnecessary computation and data movement in the actual computation. As an example, Fulcrum can be used in pattern matching to filter those parts of the text that do not match the exact part of the pattern.
- **Fulcrum with other memory technologies:** In Fulcrum, we discussed the challenges regarding implementing our method for DRAM. However, we believe the same simplified control and access mechanism can be employed for SRAM-based and NVM-based accelerators. Fulcrum with SRAM-based memory technologies can benefit from the high frequency and more efficient logic of SRAM technologies. non-volatile memories also have higher capacity. However, frequent write operations to non-volatile memories can lower endurance. Future works can evaluate Fulcrum for NVM and investigate solutions for increasing endurance. For example, our error detection mechanism proposed in the previous section could help to increase endurance.
- **Enhancing reliability and security:** Most current in-situ approaches ignore the error detection and correction mechanisms and often target only approximate applications. However, even for approximate applications, enhancing reliability and reducing error is desirable. The single-word and sequential processing of a row-buffer proposed in this dissertation may enable an early error detection mechanism for in-situ accelerators. To this end, we can assign the last column of the Walker to parity values, where each binary digit is the parity bit, for all the bits in the same digit in all columns. Consequently, adjacent bits will have different parity bits and multiple adjacent errors are still detectable. Thanks to our sequential column processing, we can xor sequentially accessed columns in each cycle. At the end of the process, the result must be equal to the last column. On the other hand, in-situ approaches are vulnerable to row hammer attacks that may flip the bit values, and the aforementioned solution for error detection can alleviate that. Moreover, the sequential processing and the three Walkers enable in-situ shuffling. The

shuffling, in addition to our xor operations can be used to implement a light-weight in-situ protection mechanism for data and enhance the security of systems.

- **Designing a subarray-level architecture that supports both row-wide bitwise operations and single-word processing units:** Recently proposed in-situ row-wide bitwise approaches (e.g., SIMDRAM [130]) implements arithmetic operations using bit-serial computation on vertically laid-out data (where each bit of a 32-bit value is in a different row). The vertical layout eliminates the extra shifts for addition operations and highly benefits few-bit integer operations. However, these approaches still are inefficient; for example, 32-bit multiplication operations require 11,103 row activations. SIMDRAM[130] also cannot support floating-point operations, because it cannot support the required shift operations. We propose to design a new in-situ approach that can exploit the benefits of row-wide bitwise for few-bit integer operations and the benefits of single-word processing for floating-point operations, 32-bit multiplication, operations with data dependency, and conditional operations.

Bibliography

[1] Animesh Jain, Parker Hill, Shih-Chieh Lin, Muneeb Khan, Md E Haque, Michael A Laurenzano, Scott Mahlke, Lingjia Tang, and Jason Mars. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *Microarchitecture , 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13, 2016.

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

[3] Kevin K Chang, Prashant J Nair, Donghyuk Lee, Saugata Ghose, Moinuddin K Qureshi, and Onur Mutlu. Low-cost inter-linked subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture* , pages 568–580, 2016.

[4] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 243–254, 2016.

[5] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom. Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods. In *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[6] Somayeh Sardashti and David A Wood. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 62–73, 2013.

[7] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 377–388, 2012.

[8] Atieh Lotfi, Abbas Rahimi, Hadi Esmaeilzadeh, and Rajesh K Gupta. SqueezCL: Squeezing OpenCL Kernels for Approximate Computing on Contemporary GPUs. In *Workshop on Approximate Computing*, 2015.

[9] Seogoo Lee, Dongwook Lee, Kyungtae Han, Emily Shriver, Lizy K John, and Andreas Gerstlauer. Statistical quality modeling of approximate hardware. In *Quality Electronic Design (ISQED), 2016 17th International Symposium on*, pages 163–168, 2016.

[10] Thierry Moreau, Felipe Augusto, Patrick Howe, Armin Alaghi, and Luis Ceze. Exploiting quality-energy tradeoffs with arbitrary quantization: special session paper. In *Proceedings of the Twelfth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis Companion*, 2017.

[11] qappa: A framework for navigating quality-energy tradeoffs with arbitrary quantization. Technical report.

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

[13] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R. Stan, and Kevin Skadron. Fulcrum: a Simplified Control and Access Mechanism toward Flexible and Practical in-situ Accelerators. In *Proceedings of the 26th IEEE International Symposium on High-Performance Computer Architecture*, 2020.

[14] Marzieh Lenjani and Kevin Skadron. Supporting moderate data dependency, position dependency, and divergence in pim-based accelerators. *IEEE Micro*, 2021.

[15] Marzieh Lenjani, Ahmed Alif, Mircea R. Stan, and Kevin Skadron. Gearbox: A Case for Supporting Accumulation Dispatching and Hybrid Partitioning in PIM-based Accelerators. In *Proceedings of the 49th International Symposium on Computer Architecture (ISCA)*. IEEE, 2022.

[16] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. Bonsai: High-performance adaptive merge tree sorting. In *47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[17] Zheyu Li, Nagadastagiri Challapalle, Akshay Krishna Ramanathan, and Vijaykrishnan Narayanan. IMC-Sort: In-Memory Parallel Sorting Architecture using Hybrid Memory Cube. In *Great Lakes Symposium on VLSI (GLSVLSI)*, 2020.

[18] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, M Arif Rahman, and Mircea R. Stan. An Overflow-free Quantized Memory Hierarchy in General-purpose Processors. In *Proceedings of the IEEE International Symposium on Workload Characterization*, 2019.

[19] Ramyad Hadidi, Bahar Asgari, Burhan Ahmad Mudassar, Saibal Mukhopadhyay, Sudhakar Yalamanchili, and Hyesoon Kim. Demystifying the characteristics of 3d-stacked memories: A case study for hybrid memory cube. In *Proceedings of the IEEE International Symposium on Workload Characterization* , pages 66–75, 2017.

[20] Amin Farmahini-Farahani, Sudhanva Gurumurthi, Gabriel Loh, and Michael Ignatowski. Challenges of high-capacity dram stacks and potential directions. In *Proceedings of the Workshop on Memory Centric High Performance Computing*, pages 4–13, 2018.

[21] Donghyuk Lee, Saugata Ghose, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Simultaneous multi-layer access: Improving 3d-stacked memory bandwidth at low cost. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):1–29, 2016.

[22] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute caches. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 481–492, 2017.

[23] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, et al. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–197, 2013.

[24] Vivek Seshadri, Kevin Hsieh, Amirali Boroum, Donghyuk Lee, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Fast bulk bitwise AND and OR in DRAM. *IEEE Computer Architecture Letters*, pages 127–131, 2015.

[25] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference*, page 173, 2016.

[26] Quan Deng, Lei Jiang, Youtao Zhang, Minxuan Zhang, and Jun Yang. DrAcc: a DRAM based accelerator for accurate CNN inference. In *Proceedings of the 55th Annual Design Automation Conference*, page 168, 2018.

[27] Mahdi Nazm Bojnordi and Engin Ipek. Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 1–13, 2016.

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

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

[30] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. FloatPIM: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[31] 3D XPoint Technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[32] Minxuan Zhou, Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. Gram: graph processing in a reram-based computational memory. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019.

[33] David Patterson, Krste Asanovic, Aaron Brown, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Christoforos Kozyrakis, David Martin, Stylianos Perissakis, et al. Intelligent ram (IRAM): the industrial setting, applications, and architectures. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, pages 2–7, 1997.

[34] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM. *IEEE MICRO*, pages 34–44, 1997.

[35] Justin Hensley, Mark Oskin, Diana Keen, Lucian-Vlad Lita, and Frederic T Chong. Active page architectures for media processing. In *First Workshop on Media Processors and DSPs, 32nd Annual Symposium on Microarchitecture*, 1999.

[36] Mark Oskin, Diana Keen, Justin Hensley, L-V Lita, and Frederic T Chong. Reducing cost and tolerating defects in page-based intelligent memory. In *Proceedings 2000 International Conference on Computer Design*, pages 276–284, 2000.

[37] Christoforos E Kozyrakis and David A Patterson. Scalable, vector processors for embedded systems. *IEEE Micro*, pages 36–45, 2003.

[38] Ngeci Bowman, Neal Cardwell, Chris Kozyrakis, Cynthia Romer, and Helen Wang. Evaluation of existing architectures in IRAM systems. In *Workshop on Mixing Logic and DRAM, 24th International Symposium on Computer Architecture*, page 23, 1997.

[39] Andreas Nowatzyk, Fong Pong, and Ashley Saulsbury. Missing the memory wall: The case for processor/memory integration. In *23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pages 90–90. IEEE, 1996.

[40] UPMEM. https://www.upmem.com/.

[41] Gokcen Kestor, Roberto Gioiosa, Darren J Kerbyson, and Adolfy Hoisie. Quantifying the energy cost of data movement in scientific applications. In *Workload Characterization , 2013 IEEE International Symposium on*, pages 56–65, 2013.

[42] Dhinakaran Pandiyan and Carole-Jean Wu. Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms. In *Workload Characterization , 2014 IEEE International Symposium on*, pages 171–180, 2014.

[43] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, et al. Google workloads for consumer devices: mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[44] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. A scalable and efficient in-memory interconnect architecture for automata processing. *IEEE Computer Architecture Letters*, 2019.

[45] Ismail Akturk and Ulya R Karpuzcu. AMNESIAC: Amnesic Automatic Computer. In *22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[46] Marzieh Lenjani and Mahmoud Reza Hashemi. A novel arbitration scheme for bandwidth and jitter guarantees in asynchronous NoCs. In *2009 14th International CSI Computer Conference*, pages 231–235. IEEE, 2009.

[47] Marzieh Lenjani and Mahmoud Reza Hashemi. Tree-based scheme for reducing shared cache miss rate leveraging regional, statistical and temporal similarities. *IET Computers & Digital Techniques*, 8(1):30–48, 2014.

[48] Marc Henneaux and Claudio Teitelboim. *Quantization of gauge systems*. 1994.

[49] Robert M Gray. Vector quantization. In *Readings in speech recognition*, pages 75–100. 1990.

[50] Allen Gersho and Robert M Gray. *Vector quantization and signal compression*. 2012.

[51] Verilog implementation of quantizer and de-quantizer modules . https://github.com/quantizerdequantizer/QuantizerDeQuantizer.

[52] NYSE Stock Exchange. http://online.wsj.com/mdc/public/page/2_3024-NYSE.html.

[53] Nacereddine Hammami and Mouldi Bedda. Improved tree model for arabic speech recognition. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, pages 521–526, 2010.

[54] Spoken Arabic Digit Data Set . https://archive.ics.uci.edu/ml/datasets/Spoken+Arabic+Digit.

[55] EEG Eye State Data Set. https://archive.ics.uci.edu/ml/datasets/EEG+Eye+State.

[56] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[57] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

[58] Amir Yazdanbakhsh, Divya Mahajan, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. AXBENCH: A Multi-Platform Benchmark Suite for Approximate Computing. *IEEE Design & Test*, 2016.

[59] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460, 2012.

[60] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–184, 2013.

[61] Credit Card Fraud Detection . https://www.kaggle.com/dalpozz/creditcardfraud.

[62] Air Quality Data Set . https://archive.ics.uci.edu/ml/datasets/Air+quality.

[63] Forest Fires Dataset . http://www3.dsi.uminho.pt/pcortez/forestfires/.

[64] Parkinson Speech Dataset with Multiple Types of Sound Recordings Data Set . https://archive.ics.uci.edu/ml/datasets/Parkinson+Speech+Dataset+with++Multiple+Types+of+Sound+Recordings.

[65] Kdd Cup 99 dataset . http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

[66] Energy Efficiency dataset . http://archive.ics.uci.edu/ml/datasets/energy+efficiency.

[67] Sales Transactions Dataset Weekly Data Set . https://archive.ics.uci.edu/ml/datasets/Sales_Transactions_Dataset_Weekly.

[68] Jason W Osborne and Amy Overbay. The power of outliers (and why researchers should always check for them). *Practical assessment, research & evaluation*, pages 1–12, 2004.

[69] Vic Barnett, Toby Lewis, et al. *Outliers in statistical data*. 1994.

[70] Jan Van den Broeck, Solveig Argeseanu Cunningham, Roger Eeckels, and Kobus Herbst. Data cleaning: detecting, diagnosing, and editing data abnormalities. *PLoS medicine*, page e267, 2005.

[71] Hancong Liu, Sirish Shah, and Wei Jiang. On-line outlier detection and data cleaning. *Computers & chemical engineering*, pages 1635–1647, 2004.

[72] Amir Roth and Gurindar S Sohi. Effective jump-pointer prefetching for linked data structures. In *ACM SIGARCH Computer Architecture News*, volume 27, pages 111–121. IEEE Computer Society, 1999.

[73] Chi-Keung Luk and Todd C Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers*, 48(2):134–141, 1999.

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

[75] Jeremy Kepner, David Bade, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. Graphs, matrices, and the GraphBLAS: Seven good reasons. *arXiv preprint arXiv:1504.01039*, 2015.

[76] Animesh Jain, Parker Hill, Michael A Laurenzano, Md E Haque, Muneeb Khan, Scott Mahlke, Lingjia Tang, and Jason Mars. CPSA: Compute Precisely Store Approximately. In *Workshop on Approximate Computing Across the Stack*, 2016.

[77] Intel 64 and IA-32 Architectures Developerś Manual . https://https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.

[78] Full description of metadata . https://github.com/quantizerdequantizer/quantizerdequantizer/blob/master/Full_description_of_metadata/.

[79] Emery David Berger. *Memory management for high-performance applications*. PhD thesis, University of Texas at Austin, 2002.

[80] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 364–373, 1990.

[81] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The Gem5 simulator. *ACM SIGARCH Computer Architecture News*, pages 1–7, 2011.

[82] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and many-core architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480, 2009.

[83] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

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

[85] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[86] James E Fowler. QccPack: An open-source software library for quantization, compression, and coding. In *Applications of digital image processing xxiii*, 2000.

[87] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. SAGE: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 13–24, 2013.

[88] Comparing Fixed- and Floating-Point DSPs . http://www.ti.com/lit/wp/spry061/spry061.pdf.

[89] Fixed-Point Digital Signal Processors . http://www.ti.com/lit/ds/symlink/tms320c6413.pdf.

[90] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. Doppelgänger: a cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 50–61, 2015.

[91] Joshua San Miguel, Jorge Albericio, Natalie Enright Jerger, and Aamer Jaleel. The bunker cache for spatio-value approximation. In *Microarchitecture , 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12, 2016.

[92] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2012.

[93] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, pages 158–169, 2016.

[94] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. TOP-PIM: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, pages 85–98, 2014.

[95] Shuangchen Li, Alvin Oliver Glova, Xing Hu, Peng Gu, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. SCOPE: A stochastic computing engine for dram-based in-situ accelerator. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* , pages 696–709, 2018.

[96] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems. In *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[97] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. In-Memory Data Parallel Processor. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–14, 2018.

[98] Ytong-Bin Kim and Tom W Chen. Assessing merged dram/logic technology. *Integration*, 27(2):179–194, 1999.

[99] Compute Express Link. https://www.computeexpresslink.org/.

[100] Tao Zhang, Ke Chen, Cong Xu, Guangyu Sun, Tao Wang, and Yuan Xie. Half-DRAM: a high-bandwidth and low-power DRAM architecture from the rethinking of fine-grained activation. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture* , pages 349–360, 2014.

[101] CACTI-3DD: Architecture-level modeling for 3d die-stacked dram main memory.

[102] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. Tiered-latency dram: A low latency and low cost dram architecture. In *International Symposium on High Performance Computer Architecture*, 2013.

[103] FULCRUM ALPU. https://github.com/MarziehLenjani/FULCRUM_ALPU.

[104] Shuai Che, Jeremy W Sheaffer, and Kevin Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 13. ACM, 2011.

[105] cuBLAS. https://docs.nvidia.com/cuda/cublas/index.html.

[106] cuSPARSE. https://docs.nvidia.com/cuda/cusparse/index.html.

[107] Thrust. https://docs.nvidia.com/cuda/thrust/index.html.

[108] Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[109] A benchmark suit for In-situ computing. https://github.com/MarziehLenjani/InSituBench.

[110] CUDA Code Samples. https://developer.nvidia.com/cuda-code-samples.

[111] knn. https://github.com/vincentfpgarcia/kNN-CUDA.

[112] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using GPU. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6, 2008.

[113] DBMS Bitmap Indexing. https://www.geeksforgeeks.org/dbms-bitmap-indexing/.

[114] Bitmap Indexes. https://docs.oracle.com/cd/A87860_01/doc/server.817/a76994/indexes.htm.

[115] Bitmap Index vs. B-tree Index: Which and When? https://www.oracle.com/technetwork/articles/sharma-indexes-093638.html.

[116] Kesheng Wu. FastBit: An efficient indexing technology for accelerating data-intensive science. In *Journal of Physics: Conference Series*, page 556, 2005.

[117] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, pages 137–148, 2013.

[118] Jeremy Appleyard. Optimizing recurrent neural networks in cudnn 5. https://devblogs.nvidia.com/optimizing-recurrent-neural-networks-cudnn-5/.

[119] MoveProf: integrating NVProf and GPUWattch for extracting the energy cost of data movement. https://github.com/MarziehLenjani/MoveProf.

[120] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 487–498, 2013.

[121] Data Sheet: Tesla P100. https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf.

[122] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. Sparse computation data dependence simplification for efficient compiler-generated inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[123] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019.

[124] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. eAP: A Scalable and Efficient In-Memory Accelerator for Automata Processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 87–99. ACM, 2019.

[125] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching. In *Proceedings of The 26th IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2020.

[126] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. FlexAmata: A universal and efficient adaption of applications to spatial automata processing accelerators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2020.

[127] Elaheh Sadredini, Reza Rahimi, Mohsen Imani, and Kevin Skadron. Sunder: Enabling low-overhead and scalable near-data pattern matching acceleration. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 311–323, 2021.

[128] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching. In *2020 IEEE international symposium on high performance computer architecture (HPCA)*, pages 86–98. IEEE, 2020.

[129] Elaheh Sadredini, Reza Rahimi, and Kevin Skadron. Enabling in-sram pattern processing with low-overhead reporting architecture. *IEEE Computer Architecture Letters*, 19(2):167–170, 2020.

[130] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur 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)*, 2021.

[131] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, et al. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology. In *Proceedings of the 48nd Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[132] Nagadastagiri Challapalle, Sahithi Rampalli, Linghao Song, Nandhini Chandramoorthy, Karthik Swaminathan, John Sampson, Yiran Chen, and Vijaykrishnan Narayanan. Gaas-x: graph analytics accelerator supporting sparse data representation using crossbar architectures. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[133] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C Hoe, Larry Pileggi, and Franz Franchetti. Efficient SPMV operation for large and highly sparse matrices using scalable multi-way merge parallelization. In *International Symposium on Microarchitecture*, 2019.

[134] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[135] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and TN 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)*, pages 372–385. IEEE, 2020.

[136] Ariful Azad and Aydin Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 688–697. IEEE, 2017.

[137] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016.

[138] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas. *ICCAD*, 2021.

[139] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the graphblas. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2016.

[140] Eriko Nurvitadhi, Asit Mishra, Yu Wang, Ganesh Venkatesh, and Debbie Marr. Hardware accelerator for analytics of sparse data. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1616–1621. IEEE, 2016.

[141] Carl Yang, Yangzihao Wang, and John D Owens. Fast sparse matrix and sparse vector multiplication algorithm on the gpu. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 841–847. IEEE, 2015.

[142] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. Thundergp: Hls-based graph processing framework on fpgas. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 69–80, 2021.

[143] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages*, 2:1–30, 2018.

[144] Carl Yang, Aydin Buluc, and John D Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu. *arXiv preprint arXiv:1908.01407*, 2019.

[145] Shijie Zhou, Rajgopal Kannan, Viktor K Prasanna, Guna Seetharaman, and Qing Wu. Hitgraph: High-throughput graph processing framework on fpga. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2249–2264, 2019.

[146] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[147] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, and Hai Jin. A survey on graph processing accelerators: Challenges and opportunities. *Journal of Computer Science and Technology*, 34(2):339–371, 2019.

[148] Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, Hongjung Kim, Ju Young Kim, Young Jun Park, Jae Hwan Kim, Dae Suk Kim, Heat Bit Park, Jin Wook Shin, et al. 25.2 a 1.2 v 8Gb 8-channel 128gb/s high-bandwidth memory (HBM) stacked DRAM with effective microbump i/o test methods using 29nm process and tsv. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 432–433. IEEE, 2014.

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

[150] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 375–386, 2015.

[151] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

[152] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, et al. 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2 TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 350–352. IEEE, 2021.

[153] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[154] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. Nxgraph: An efficient graph processing system on a single machine. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 409–420. IEEE, 2016.

[155] Seongyun Ko and Wook-Shin Han. TurboGraph++ A scalable and fast graph analytics system. In *Proceedings of the 2018 international conference on management of data*, pages 395–410, 2018.

[156] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[157] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a {PC}. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012.

[158] NVLINK AND NVSWITCH, The Building Blocks of Advanced Multi-GPU Communication . https://www.nvidia.com/en-us/data-center/nvlink/.

[159] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[160] Collective Operations . https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html.

[161] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[162] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. Gunrock: Gpu graph analytics. *ACM Transactions on Parallel Computing (TOPC)*, 4(1):1–49, 2017.

[163] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W Keckler, and William J Dally. Fine-grained DRAM: Energy-efficient DRAM for extreme bandwidth systems. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[164] Yasuko Eckert, Nuwan Jayasena, and Gabriel H Loh. Thermal feasibility of die-stacked processing in memory. 2014.

[165] Amir Yazdanbakhsh, Choungki Song, Jacob Sacks, Pejman Lotfi-Kamran, Hadi Esmaeilzadeh, and Nam Sung Kim. In-DRAM near-data approximate acceleration for GPUs. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018.

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

[167] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. GraphH: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):640–653, 2018.

[168] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks. In *2017 IEEE International symposium on high performance computer architecture (HPCA)*, pages 457–468. IEEE, 2017.

[169] Patricia Gonzalez-Guerrero, Tommy Tracy II, Xinfei Guo, Rahul Sreekumar, Marzieh Lenjani, Kevin Skadron, and Mircea R Stan. Towards on-node machine learning for ultra-low-power sensors using asynchronous σ δ streams. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 16(4):1–20, 2020.

[170] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy efficient architecture for graph analytics accelerators. *ACM SIGARCH Computer Architecture News*, 44(3):166–177, 2016.

[171] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Foregraph: Exploring large-scale graph processing on multi-FPGA architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 217–226, 2017.

[172] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015.

[173] Elias Stehle and Hans-Arno Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. In *ACM International Conference on Management of Data*, pages 417–432, 2017.

[174] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. FANS: FPGA-Accelerated Near-Storage Sorting. In *29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021.

[175] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[176] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *nature*, 393(6684):440–442, 1998.

[177] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[178] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.

[179] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[180] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014.

[181] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Phi: Architectural support for synchronization-and bandwidth-efficient commutative scatter updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[182] Graph500 benchmark. www.graph500.org.

[183] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *IEEE International Symposium on Workload Characterization*, 2015.

[184] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. Duality cache for data parallel acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[185] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[186] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. Alrescha: A lightweight reconfigurable sparse-computation accelerator. In *Proceedings of the 26th IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2020.

[187] GraphCore. https://www.graphcore.ai/.

[188] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. yaSpMV: Yet Another SpMV Framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 107–118, 2014.

[189] Matthew Poremba, Tao Zhang, and Yuan Xie. Fine-granularity tile-level parallelism in non-volatile memory architecture with two-dimensional bank subdivision. In *Proceedings of the 53rd Annual Design Automation Conference*, 2016.

[190] Eriko Nurvitadhi, Asit Mishra, and Debbie Marr. A sparse matrix vector multiply accelerator for support vector machine. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2015.

[191] The STREAM2 Home Page. https://www.cs.virginia.edu/stream/stream2/.

[192] Qing Guo, Xiaochen Guo, Ravi Patel, Engin Ipek, and Eby G Friedman. Ac-dimm: associative computing with stt-mram. *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[193] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[194] STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[195] D Anoushe Jamshidi, Mehrzad Samadi, and Scott Mahlke. D 2 MA: accelerating coarse-grained data transfer for GPUs. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 431–442, 2014.

[196] Abhijeet Gaikwad and Ioane Muni Toke. Parallel iterative linear solvers on GPU: a financial engineering case. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 607–614, 2010.

[197] Luc Buatois, Guillaume Caumon, and Bruno Lévy. Concurrent number cruncher: An efficient sparse linear solver on the GPU. In *International Conference on High Performance Computing and Communications*, pages 358–371, 2007.

[198] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. Sparse matrix format selection with multiclass SVM for SpMV on GPU. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 496–505, 2016.

[199] Xintian Yang, Srinivasan Parthasarathy, and Ponnuswamy Sadayappan. Fast sparse matrix-vector multiplication on GPUs: implications for graph mining. *Proceedings of the VLDB Endowment*, pages 231–242, 2011.

[200] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on GPUs: A survey. *ACM Computing Surveys (CSUR)*, page 81, 2018.

[201] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84, 2017.

[202] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 806–814, 2015.

[203] Arun Subramaniyan, Jingcheng Wang, Ezhil RM Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 259–272, 2017.

[204] Tesla's Auto driving processor . https://youtu.be/Ucp0TTmvqOE?t=4381.

[205] Reza Farivar, Harshit Kharbanda, Shivaram Venkataraman, and Roy H Campbell. An algorithm for fast edit distance computation on gpus. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9, 2012.

[206] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, pages 10–14, 2014.

[207] Jochen Blom, Tobias Jakobi, Daniel Doppmeier, Sebastian Jaenicke, Jörn Kalinowski, Jens Stoye, and Alexander Goesmann. Exact and complete short-read alignment to microbial genomes using Graphics Processing Unit programming. *Bioinformatics*, pages 1351–1358, 2011.

[208] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74, 2010.

[209] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 137–151, 2019.

[210] BLAS Level 1 Routines and Functions . https://software.intel.com/en-us/mkl-developer-reference-fortran-blas-level-1-routines-and-functions.

[211] BLAS Level 2 Routines and Functions . https://software.intel.com/en-us/mkl-developer-reference-fortran-blas-level-2-routines.

[212] BLAS Level 3 Routines and Functions . https://software.intel.com/en-us/mkl-developer-reference-fortran-blas-level-3-routines.

[213] Mohsen Imani, Saransh Gupta, Atl Arredondo, and Tajana Rosing. Efficient query processing in crossbar memory. In *Low Power Electronics and Design (ISLPED, 2017 IEEE/ACM International Symposium on*, pages 1–6, 2017.

[214] Mohsen Imani, Saransh Gupta, Sahil Sharma, and Tajana Rosing. NVQuery: Efficient Query Processing in Non-Volatile Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[215] Leonid Yavits, Shahar Kvatinsky, Amir Morad, and Ran Ginosar. Resistive associative processor. *IEEE Computer Architecture Letters*, pages 148–151, 2015.

[216] Qing Guo, Xiaochen Guo, Yuxin Bai, and Engin Ipek. A resistive TCAM accelerator for data-intensive computing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 339–350, 2011.

[217] Andrew D Kent and Daniel C Worledge. A new spin on magnetic memories. *Nature nanotechnology*, page 187, 2015.

[218] Luc Thomas et al. Basic Principles, Challenges and Opportunities of STT-MRAM for Embedded Memory Applications. *MSST 2017*, 2017.

[219] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M Patel. The Case Against Specialized Graph Analytics Engines. In *CIDR*, 2015.

[220] Derrick E Wood and Steven L Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, page R46, 2014.

[221] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, pages 1735–1780, 1997.

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

[223] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1994.

[224] Firstname1 Lastname1 and Firstname2 Lastname2. A very nice paper to cite. In *Proceedings of the 33rd Annual ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[225] Firstname1 Lastname1, Firstname2 Lastname2, and Firstname3 Lastname3. Another very nice paper to cite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2011.

[226] Firstname1 Lastname1, Firstname2 Lastname2, Firstname3 Lastname3, Firstname4 Lastname4, and Firstname5 Lastname5. Yet another very nice paper to cite, with many author names all spelled out. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[227] David A Patterson. *Computer architecture: A quantitative approach*. 2011.

[228] Shekhar Borkar. Role of interconnects in the future of computing. *Journal of Lightwave Technology*, pages 3927–3933, 2013.

[229] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[230] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *44th International Symposium on Computer Architecture*, 2017.

[231] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, 2012.

[232] Altaf Abdul Gaffar, Oskar Mencer, and Wayne Luk. Unifying bit-width optimisation for fixed-point and floating-point designs. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 79–88, 2004.

[233] William G Osborne, Ray CC Cheung, José Gabriel F Coutinho, Wayne Luk, and Oskar Mencer. Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 617–620, 2007.

[234] Xi Chen, Lei Yang, Robert P Dick, Li Shang, and Haris Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *IEEE transactions on very large scale integration (VLSI) systems*, pages 1196–1208, 2010.

[235] Angelos Arelakis and Per Stenstrom. SC2: A statistical compression cache scheme. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[236] Alaa R Alameldeen and David A Wood. Frequent pattern compression: A significance-based compression scheme for L2 caches. *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, 2004.

[237] Alaa R Alameldeen and David A Wood. Adaptive cache compression for high-performance processors. In *Computer Architecture. Proceedings. 31st Annual International Symposium on*, pages 212–223, 2004.

[238] M. Lichman. UCI machine learning repository, 2013.

[239] Xiaolong Jin, Benjamin W Wah, Xueqi Cheng, and Yuanzhuo Wang. Significance and challenges of big data research. *Big Data Research*, pages 59–64, 2015.

[240] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[241] Budirijanto Purnomo, Jonathan Bilodeau, Jonathan D Cohen, and Subodh Kumar. Hardware-compatible vertex compression using quantization and simplification. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 53–61, 2005.

[242] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, Raquel Urtasun, and Andreas Moshovos. Proteus: Exploiting Precision Variability in Deep Neural Networks. *Parallel Computing*, 2017.

[243] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *ACM Transactions on Computer Systems (TOCS)*, page 9, 2014.

[244] Woongki Baek and Trishul M. Chilimbi. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[245] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[246] Cumulative Percent Distribution of Population by Height and Sex: 2007 to 2008 . https://www2.census.gov/library/publications/2010/compendia/statab/130ed/tables/11s0205.pdf.

[247] Parasoft Development Testing Platform . https://docs.parasoft.com/display/DTP533/Parasoft+Development+Testing+Platform.

[248] Robert Gray. Vector quantization. *IEEE Assp Magazine*, pages 4–29, 1984.

[249] Brian Chen and Gregory W Wornell. Quantization index modulation: A class of provably good methods for digital watermarking and information embedding. *IEEE Transactions on Information Theory*, pages 1423–1443, 2001.

[250] Arshid Syed. Tradeoffs Between Combinational and Sequential Dividers. https://www.synopsys.com/dw/dwtb.php?a=fp_dividers.

[251] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. 2013.

[252] Vijay Sathish, Michael J Schulte, and Nam Sung Kim. Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads. In *Parallel Architectures and Compilation Techniques (PACT), 2012 21st International Conference on*, pages 325–334, 2012.

[253] Daniel Citron. Exploiting low entropy to reduce wire delay. *IEEE Computer Architecture Letters*, pages 1–1, 2004.

[254] How to Manipulate Data Structure to Optimize Memory Use on 32-Bit Intel Architecture . https://software.intel.com/en-us/articles/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture.

[255] Jeff Bonwick et al. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX summer*, 1994.

[256] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. Hardware supported persistent object address translation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 800–812, 2017.

[257] Martin Burtscher and Paruj Ratanaworabhan. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*, pages 18–31, 2009.

[258] Cortex-M3 Technical Reference Manual . http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337e/BABBCJII.html.

[259] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 50, 2016.

[260] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 2012.

[261] Mingyu Gao and Christos Kozyrakis. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture* , 2016.

[262] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture* , 2015.

[263] Natalie D Enright Jerger, Li-Shiuan Peh, and Mikko H Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, 2008.

[264] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V Adve, Vikram S Adve, Nicholas P Carter, and Ching-Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, 2011.

[265] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, pages 246–257, 2005.

[266] Julien Dusser, Thomas Piquet, and André Seznec. Zero-content augmented caches. In *Proceedings of the 23rd international conference on Supercomputing*, 2009.

[267] Andreas Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, pages 234–245, 2005.

[268] Tongping Liu and Emery D. Berger. SHERIFF: Precise Detection and Automatic Mitigation of False Sharing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 3–18, 2011.

[269] amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy.

[270] Avoiding and Identifying False Sharing Among Threads . https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads.

[271] Ananthanarayanan, Ganesh and Hung, Michael Chien-Chun and Ren, Xiaoqi and Stoica, Ion and Wierman, Adam and Yu, Minlan. GRASS: trimming stragglers in approximation analytics. 2014.

[272] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[273] Interactive Big Data analysis using approximate answers . https://www.oreilly.com/ideas/interactive-big-data-analysis-using-approximate-answers.

[274] libfixmath . https://en.wikipedia.org/wiki/Libfixmath.

[275] libfixmatrix . https://github.com/PetteriAimonen/libfixmatrix.

[276] Q (number format . https://en.wikipedia.org/wiki/Q_(number_format).

[277] Santosh G. Abraham and David E. Hudak. Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic. *IEEE Transactions on Parallel and Distributed Systems*, 1991.

[278] Qing Guo, Karin Strauss, Luis Ceze, and Henrique S. Malvar. High-Density Image Storage Using Approximate Memory Cells. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 413–426, 2016.

[279] Djordje Jevdjic, Karin Strauss, Luis Ceze, and Henrique S. Malvar. Approximate Storage of Compressed and Encrypted Videos. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 361–373, 2017.

[280] Fixed-Point vs. Floating-Point Digital Signal Processing. http://www.analog.com/en/education/education-library/articles/fixed-point-vs-floating-point-dsp.html.

[281] Nandita Vijaykumar, Abhilasha Jain, Diptesh Majumdar, Kevin Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Phillip B Gibbons, and Onur Mutlu. A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture* , 2018.

[282] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B Gibbons, and Onur Mutlu. The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs. In *In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, 2018.

[283] Prasanna Venkatesh Rengasamy, Anand Sivasubramaniam, Mahmut T Kandemir, and Chita R Das. Exploiting staleness for approximating loads on CMPs. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, 2015.

[284] Blocked Matrix Multiplication . https://malithjayaweera.com/2020/07/blocked-matrix-multiplication/.

[285] Daichi Fujiki, Xiaowei Wang, Arun Subramaniyan, and Reetuparna Das. In-/near-memory computing. *Synthesis Lectures on Computer Architecture*, 16(2):1–140, 2021.

[286] Xinfeng Xie, Peng Gu, Yufei Ding, Dimin Niu, Hongzhong Zheng, and Yuan Xie. MPU: Towards Bandwidth-abundant SIMT Processor via Near-bank Computing. *arXiv preprint arXiv:2103.06653*, 2021.

[287] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. *arXiv preprint arXiv:2105.03814*, 2021.

[288] Peng Gu, Xinfeng Xie, Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, and Yuan Xie. iPIM: Programmable in-memory image processing accelerator using near-bank architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 804–817. IEEE, 2020.

[289] Lingxi Wu, Rasool Sharifi, Marzieh Lenjani, Kevin Skadron, and Ashish Venkat. Sieve: Scalable In-situ DRAM-based Accelerator Designs for Massively Parallel k-mer Matching. 2021.

[290] Decadal Plan for Semiconductors. https://www.src.org/about/decadal-plan/.

[291] A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development. https://www.acm.org/hennessy-patterson-turing-lecture.

[292] Lotfi Belkhir and Ahmed Elmeligi. Assessing ICT global emissions footprint: Trends to 2040 & recommendations. *Journal of cleaner production*, 177:448–463, 2018.

[293] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *2012 symposium on VLSI technology (VLSIT)*, pages 87–88. IEEE, 2012.

[294] Tayo Oguntebi and Kunle Olukotun. Graphops: A dataflow library for graph analytics acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 111–117, 2016.

[295] Shaahin Angizi and Deliang Fan. GraphiDe: A graph processing accelerator leveraging in-DRAM-computing. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pages 45–50, 2019.

[296] Kyomin Sohn, Won-Joo Yun, Reum Oh, Chi-Sung Oh, Seong-Young Seo, Min-Sang Park, Dong-Hak Shin, Won-Chang Jung, Sang-Hoon Shin, Je-Min Ryu, et al. A 1.2 V 20 nm 307 GB/s HBM DRAM with at-speed wafer-level IO test scheme and adaptive refresh considering temperature distribution. *IEEE Journal of Solid-State Circuits*, 52(1):250–260, 2016.

[297] Chad D Kersey, Hyesoon Kim, and Sudhakar Yalamanchili. Lightweight SIMT core designs for intelligent 3D stacked DRAM. In *Proceedings of the International Symposium on Memory Systems*, pages 49–59, 2017.

[298] waferScale. https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-lear

[299] Jun Liu, Jagadish Kotra, Wei Ding, and Mahmut Kandemir. Network footprint reduction through data access and computation placement in noc-based manycores. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[300] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. Toward standardized near-data processing with unrestricted data placement for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[301] Application task and data placement in embedded many-core numa architectures.

[302] Jagadish B Kotra, Seongbeom Kim, Kamesh Madduri, and Mahmut T Kandemir. Congestion-aware memory management on numa platforms: A vmware esxi case study. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 146–155. IEEE, 2017.

[303] Yuho Jin. Unifying router power gating with data placement for energy-efficient noc. In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 74–81. IEEE, 2015.

[304] Deferred updates and data placement in distributed databases.

[305] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Habinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. 2010.

[306] Ann Chervenak, Ewa Deelman, Miron Livny, Mei-Hui Su, Rob Schuler, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Data placement for scientific applications in distributed environments. In *2007 8th IEEE/ACM International Conference on Grid Computing*, pages 267–274. IEEE, 2007.

[307] André Brinkmann, Kay Salzwedel, and Christian Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 119–128, 2000.

[308] DLRM: An advanced, open source deep learning recommendation model. https://ai.facebook.com/blog/dlrm-an-advanced-open-source-deep-learning-recommendation-model/.

[309] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[310] Aydın Buluç and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[311] William J Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Communications of the ACM*, 63(7):48–57, 2020.