

**EFFICIENTLY ACCELERATING SPARSE PROBLEMS BY ENABLING
STREAM ACCESSES TO MEMORY USING HARDWARE/SOFTWARE
TECHNIQUES**

A Dissertation
Presented to
The Academic Faculty

By

Bahar Asgari

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
Electrical and Computer Engineering
Department of Engineering

Georgia Institute of Technology

May 2021

© Bahar Asgari 2021

**EFFICIENTLY ACCELERATING SPARSE PROBLEMS BY ENABLING
STREAM ACCESSES TO MEMORY USING HARDWARE/SOFTWARE
TECHNIQUES**

Thesis committee:

Dr. Hyesoon Kim
Computer Science
Georgia Institute of Technology

Dr. Richard Vuduc
Computer Science
Georgia Institute of Technology

Dr. Saibal Mukhopadhyay
Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Tushar Krishna
Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Moinuddin Qureshi
Computer Science
Georgia Institute of Technology

Date approved: April 26, 2021

Let the beauty of what you love be what you do.

Rumi

To my family.

ACKNOWLEDGMENTS

Foremost, I would like to express my deepest appreciation to my late advisor Dr. Sudhakar Yalamanchili. I will be ever grateful for his assistance. His memory will be with me always. I would also like to extend my deepest gratitude to my advisor Dr. Hyesoon Kim for her profound belief in my abilities and my work and her unrelenting support. The completion of my dissertation would not have been possible without her support and nurturing.

I would like to extend my gratitude to the members of my thesis committee for their help in the preparation of this work – Dr. Saibal Mukhopadhyay for his valuable advice that had no small part to play in the formation of this dissertation, Dr. Tushar Krishna for always listening to my ideas and suggesting ingenious insights for improving them, Dr. Moinuddin Qureshi for his unwavering support, advice, and encouragement at any time throughout the duration of my Ph.D. studies, and Dr. Richard Vuduc for his constructive advice and practical suggestions. I am also extremely grateful to Dr. Sung-Kyu Lim whose help and contributions to this work cannot be overestimated.

I would like to gratefully acknowledge the support of the Laboratory for Physical Sciences (LPS) and extend my sincere thanks to Dr. Eric Cheng from LPS who provided insight and expertise that greatly assisted this research. I had the great pleasure of collaborating with Dr. Ramyad Hadidi, Dr. Prashant Nair, Dr. Jeffrey Young, Jiashen Cao, Da Eun Shim, Dheeraj Ramchandani, Amaan Marfatia, Nima Shoghi, Sam Jijina, Adriana Amyette, Fares Elsabbagh, Younmin Bae, Burhan Ahmad Mudassar, and Kartikay Garg, without whom I would have never completed the pieces of this work. I also would like to extend my special thanks to Dr. Fernando Mujica for his continuous support and encouragement. I cannot leave Georgia Tech without mentioning Daniela Staiculescu, who was always there to help. Thanks should also go to Jane Chisholm without whose guidance I could have never written this dissertation.

I wish to thank the former members of CASL lab, Dr. Xinwei Chen, Dr. Karthik

Rao, Dr. Chad Kersey, and Dr. He Xiao who always provided me with encouragement and enlivening. I also like to extend my gratitude to the former and current members of the HPArch lab, Dr. Hyojong Kim, Dr. Prasun Gera, Yonghae Kim, Jaewon Lee, Andrei Bersatti, Blaise Tine, and Euna Kim. I was lucky to know them during my years at Georgia Tech and have fruitful discussions with them. Many thanks are due to all my friends who made the journey of my Ph.D. most wonderful.

Above all, I am deeply indebted to my family for their unparalleled love and support. I cannot begin to express my thanks to my parents Vida Zehforoosh and Aziz Asgari who selflessly encouraged me to explore new directions in life and seek my own destiny. Last but not least, I am extremely grateful to my sister, Shadi Asgari for always being there for me as the best friend.

TABLE OF CONTENTS

Acknowledgments	v
List of Tables	xi
List of Figures	xiii
Summary	xxi
Chapter 1: Introduction and Background	1
1.1 Sparse Problems	2
1.2 Contributions	4
1.3 Thesis Statement	9
Chapter 2: Literature Survey and the Gap of Knowledge	10
2.1 Specialized Hardware for Sparse Acceleration	10
2.1.1 Near-Data Processing for Recommendation Systems	10
2.1.2 Hardware Accelerators for Neural Networks	15
2.1.3 Hardware Accelerators for Sparse Matrix Algebra and Scientific Computing	17
2.2 Software Techniques for Dealing with Sparsity	19
2.2.1 Pruning the Neural Networks	19

2.2.2	Compressing Sparse Matrices	21
2.3	Unsolved Challenges	24
2.3.1	The Underutilization of Memory Bandwidth	25
2.3.2	The Underutilization of Dense Computation	29
Chapter 3: Processing Sparse Data While Gathering Them		31
3.1	Main Contributions of Fafnir	31
3.2	The Top-Down Overview of Fafnir	33
3.3	Key Mechanisms	38
3.4	Adapting Fafnir to SpMV	40
3.5	Evaluation	43
3.5.1	Experimental Setup	43
3.5.2	Latency	46
3.5.3	Speedup and Scalability	47
3.5.4	Power Consumption and Area Overhead	49
3.6	Summary	52
Chapter 4: Mathematical Transformation to Reduce Dependencies		53
4.1	Key Mechanisms of Alrescha	53
4.2	Compression Format for Sparse PDEs	56
4.3	Broad Applications	57
4.4	Putting Them Together for Sparse PDEs	60
4.5	Reconfigurable Microarchitecture	63
4.6	Evaluation	64

4.6.1	Experimental Setup	64
4.6.2	Execution Time	66
4.6.3	Energy Consumption	70
4.6.4	FPGA Resource Utilization and Power Consumption	71
4.7	Summary	72
Chapter 5: Structured Pruning for Systolic Arrays		74
5.1	Lodestar Pruning Algorithm	74
5.2	Eridanus Systolic Microarchitecture	76
5.3	Evaluation	78
5.3.1	Experimental Setup	78
5.3.2	Accuracy	79
5.3.3	Performance	79
5.4	Summary	81
Chapter 6: Fast Decompression		83
6.1	Ascella Decompression Mechanism	84
6.2	Evaluation	85
6.2.1	Experimental setup	85
6.2.2	The Overhead of Decompression	94
6.2.3	Latency and Balance Ratio	96
6.2.4	Throughput and Bandwidth Utilization	98
6.2.5	Resource Utilization and Power Consumption	100
6.3	Summary	102

Chapter 7: Conclusions	103
References	106

LIST OF TABLES

1.1	The challenges and applications of sparse problems, our contributions and publications to resolve the challenges, and the broader impacts that this research would have.	5
2.1	State-of-the-art approaches for accelerating graph and scientific problems . .	19
2.2	The impact of pruning granularity on inference using systolic arrays.	21
3.1	FIFO buffer sizes that are sum of all buffers in all PEs (B is batch size). . .	37
3.2	SpMV vs. embedding lookup	42
3.3	Sparse matrices from SuiteSparse [117].	45
3.4	Latency (cycles @200MHz) of the components in compute units of Fafnir for FPGA implementation.	46
3.5	FPGA resource utilization for Fafnir.	50
3.6	Area and power consumption breakdown @500MHz to switching (Sw.), interconnections (Int.), and leakage (Lkg.) for ASIC design of Fafnir @7 nm.	51
4.1	The properties of sparse kernels and corresponding dense data paths, implemented in Alrescha. Depending on the type of kernel, the <i>operation</i> in phase 1 can use the three vector operands at the same time or use just two of them.	59
4.2	Baseline configurations.	65
4.3	Alrescha Configuration.	66
4.4	Resource utilization and the total dynamic power consumption.	72

6.1	Resource utilization and the total dynamic power consumption for three partition sizes (8, 16, and 32).	101
-----	--	-----

LIST OF FIGURES

1.1	Memory and computation latency for streaming blocks for (a) prior work, and (b) Ascella. Closer memory and compute is better	9
2.1	Comparing NDP-based solutions for embedding lookup: (a) Baseline with no NDP, (b) TensorDIMM [35], and (c) RecNMP [36].	13
2.2	<i>Sparse formats:</i> (a) The original sparse matrix in dense format, (b) CSR including offsets to indicate the number of non-zero entries per row, indices, indicating column indices of non-zero entries, and the values itself. CSC follows the same rule as CSR; (c) BCSR including offsets to indicate the number of non-zero 4×4 blocks per row, indices, indicating the index of the first column of non-zero blocks, and the values, indicating the flatten values of non-zero blocks; (d) COO including a series of (row, column, value) tuples for non-zero values; (e) DOK , which is similar to COO; (f) LIL , which pushes all the non-zero entries to top and saves the row indices; (g) ELL , which is similar to LIL but pushes the non-zero entries to left and also uses a padding; and (h) DIA , which saves the non-zero diagonals by adding the diagonal numbers as a header to each diagonal.	23
2.3	The time to perform embedding lookup in recommendation systems when the size of memory system grows. Numbers are normalized to a one-rank system.	25
2.4	Dependencies in SymGS: Each iteration of the outer loop reads the entire vector x (left) and updates one element of x (right). The iterations of the outer loop are dependent.	26
2.5	Limited parallelism: Unrolling the iterations of the outer loop (left) and mapping the parallelizable iterations of the inner loop to processing units of a GPU (right).	27

2.6	(a) Key observation: the iterations of the outer loop are just <i>partially</i> dependent. In fact, only a few iterations of the inner loop read the newly updated elements. Therefore, we can break down the iterations of the inner loop across a few unrolled iterations of the outer loop, into data-dependent part ($j = 4, 5, 6$), and parallelizable part (green parts); (b) Key challenge: ineffectiveness of blocking technique on GPUs. More parallelism at step 1, but dependencies still create the bottleneck through steps 2, 3, and 4.	28
2.7	The time steps required to read a sparse matrix compressed in CSR and BCSR formats, for creating the non-zero rows. For simplicity, this example (and the next example in Figure 6.1) assume only one read per cycle from a buffer.	29
2.8	Clarifying the challenge of systolic-array underutilization: Comparing the execution of a matrix-matrix multiplication using a systolic array when one operand is a dense matrix and the other operand is: (a) a sparse matrix, and (b) a locally-dense sparse matrix.	30
3.1	The percentage of unique indices in batches of queries.	33
3.2	(a) The architecture of Fafnir tree, consisting of DIMM/rank and channel nodes and ASIC designs at 7 nm for a PE and a DIMM/rank node. (b) The mapping of embedding tables to memory addresses.	35
3.3	The microarchitecture of a PE including FIFO buffers, compute, and merge units, showing the data path from leaves to the root.	36
3.4	Concurrent batch processing and eliminating redundant memory accesses in Fafnir: (a) A batch of four queries that access random embedding vectors from eight embedding tables and a three-level Fafnir tree (b) Extracting the unique indices of four queries and creating the headers of requests to be forwarded to Fafnir. The steps of processing the four queries through the PEs at three levels of tree: (c) L0, (d) L1, and (e) L2.	38
3.5	(a) Embedding lookup in Fafnir, (b) Using Fafnir for an SpMV with no mechanisms, and (c) Using vectorization to fully utilize Fafnir for SpMV.	41
3.6	The iterations and rounds for SpMV on large sparse matrices using Fafnir when only n columns of the matrix fits to Fafnir at a time.	42
3.7	The number of iterations, rounds per iteration, and required merges for matrices with up to 20 million columns, for vector sizes (a) 1024 and (b) 2048. In our configuration for SpMV, vector size (i.e., the number of columns that fit in Fafnir tree) is 2048.	43

3.8	Experimental setup.	44
3.9	Single-query latency breakdown.	47
3.10	End-to-end inference speedup for DLRM on Kaggle (batch size = 8): (a) RecNMP, (b) Fafnir.	48
3.11	Speedup of Fafnir over Two-Step algorithm [76] for two SpMV-based applications: scientific computations (matrix inversion algorithm) and graph.	48
3.12	Speedup of Fafnir and TensorDIMM [35] (TDM) over RecNMP [36] (RNMP) for batch sizes (a) 8, (b) 16, and (c) 32. Opt. stands for the optimization of elimination of the redundant memory accesses.	49
3.13	(a) Dynamic power consumption breakdown of Fafnir on FPGA. (b) Power distribution of a PE in our ASIC design at 7 nm.	50
3.14	Number of memory accesses at different DIMMs for three batch sizes:(a) 8, (b) 16, and (c) 32.	51
4.1	Key insight of Alrescha: We divide a large SymGS into a majority of parallelizable GEMV operations (green) and a minority of <i>small</i> data-dependent SymGS (pink). We first run the GEMV and then switch to SymGS. Dependencies still exist in SymGS part but as long as it is small, we can run them in one step in hardware rather than three steps (❶). Forwarding the outcomes of GEMV to the SyMGS must be fast as well (❷).	54
4.2	Key mechanisms of Alrescha: (a) Matrix A the operand of the original large SymGS, (b, c, d) Executing GEMV and the mechanism for quickly forwarding the outcome of GEMV to SymGS using a LIFO or FIFO; and (e, f, g) Executing the small SymGS and implementing the scheme of data dependencies through the interconnections between inputs of the tree and the LIFO to quickly execute the small SymGS.	55
4.3	Compression format of Alrescha: the <code>col_index</code> of BCSR and <code>input_index</code> (i.e., Inx_{in}) of Alrescha are color-coded to show their corresponding blocks in the matrix. Alrescha uses the index of the last column for the input index of diagonal blocks.	58
4.4	The overview of Alrescha and host.	60
4.5	The order of operations: An example of the configuration table for a SymGS kernel, in which $n = 9$, $\omega = 3$	62

4.6	The microarchitecture of Alrescha: (a) the FCU for implementing common computations, and the RCU for providing specific configuration for distinct dense data paths. Example configurations for supporting: (b) D-SymGS, (c) GEMV, and (d) D-PR.	63
4.7	Speedup: PCG algorithm on scientific datasets, normalized to GPU (bar charts), and bandwidth utilization (the lines) compared to the state-of-the-art accelerator for scientific problems [73].	67
4.8	Reduction in data-dependent operations: for the PCG algorithm, after applying Alrescha. The baseline shows the percentage of data-dependent operations by row-reordering optimization.	68
4.9	Speedup: graph algorithms on graph datasets over the CPU. GraphR [72] is the state-of-the-art graph accelerator.	69
4.10	Speedup: executing SpMV on scientific and graph datasets normalized to GPU (bar charts), and the percentage of execution time devoted to cache accesses (the lines). OuterSPACE [69] is the state-of-the-art SpMV accelerator.	70
4.11	Energy consumption of Alrescha normalized to CPU and GPU.	71
5.1	An overview of the systolic-based microarchitecture of Eridanus.	76
5.2	The results of implementing Lodestar on Eridanus: (a) The accuracy and the percentage of zero blocks for CifarNet (pruned between step 20k, 100k) and VGG16 (pruned between steps 1, 10k). (b) Throughput and bandwidth utilization. (c) the power efficiency of three DNN models pruned by various structured techniques.	79
6.1	(a) Compressing a sparse matrix using LIL storage. (b) Time steps of reading indices and values to decompress the non-zero rows.	84
6.2	<i>The architecture of our evaluation platform:</i> Streaming the compressed partitions of a sparse matrix from the memory to FPGA through AXI stream interfaces and processing them (i.e., SpMV) in a pipeline. The <i>decompression</i> component varies based on sparse format.	85
6.3	The microarchitecture of Ascella for decompression.	86
6.4	The steps of decompression using Ascella.	86

6.5	Decompression overhead for SuiteSparse: comparing the latency overhead, σ (lower is better), of seven sparse formats for partition size of 16×16 . A darker color indicates less sparsity (i.e., higher density).	94
6.6	Decompression overhead for random matrices: comparing the σ (lower is better) of seven formats for 16×16 partitions when density varies from 0.0001 to 0.5.	95
6.7	Decompression overhead for band matrices: comparing the latency overhead, σ , of seven sparse formats for partition size of 16×16 when the width varies from 1 to 64.	95
6.8	Decompression overhead for various partition sizes: comparing the average σ (lower is better) of seven sparse formats for three types of workloads (SuiteSparse, random, band) and partition sizes of 8, 16, and 32.	96
6.9	Balance ratio: the relationship between the memory and compute latency for various partition sizes indicated by the size of markers for (a) SuiteSparse, (b) random workloads, and (c) band matrices. The blue line indicates balance ratio = 1.	97
6.10	Throughput vs. latency: comparing the throughput and the average time to apply SpMV on an 8000×8000 : (a) CSR, (b) BCSR, (c) COO, (d) LIL, (e) ELL, (f) DIA, and (g) CSC. Thicker lines indicate larger partition sizes. . . .	99
6.11	Memory bandwidth utilization for random matrices: comparing seven sparse formats for partition size of 16×16 when density varies from 0.0001 to 0.5. . . .	99
6.12	Memory bandwidth utilization for band matrices: comparing seven sparse formats for partition size of 16×16 when the width varies from 1 to 64. . . .	100
6.13	<i>Memory bandwidth utilization for various partition sizes</i> : comparing the average memory bandwidth utilization (higher is better) on SuiteSparse, random, band workloads for partition sizes of 8, 16, and 32.	100
6.14	Dynamic power consumption: (a) logic, (b) BRAM, and (c) signals.	102
7.1	Comparing sparse formats for: (a) SuiteSparse, (b) random, and (c) band matrices. 1 and 0 show best and worst, respectively.	104

SUMMARY

Approaching the end of Moore’s law has conveyed the computer architecture toward specialized hardware accelerators that have become more attractive because of the recent growing demand of compute- and memory-intensive algorithms with a high level of parallelism and/or specific patterns of data reuse (e.g., machine learning, scientific computing, and graph analytics.) The performance gained by the migration from CPUs to GPUs, FPGAs, and ASIC has been obtained thanks to two main reasons: (i) tailoring the hardware for the requirements of the applications. This reason is more observable in the parallelizable applications such as the inference of neural networks, the performance of which can be significantly improved by heavily concurrent processors that can process hundreds of thousands of operations at a cycle (e.g. TPU); and (ii) technology scaling and thereby enabling higher clock frequency and lower switching delay. The performance gained by the implementation of a given algorithm on FPGA or ASIC is due to the second fact. Although designing hardware accelerators based on the two preceding facts has shown significant interests, that might encounter certain performance scaling limitations if they still depend on technology scaling. *Independency from technology scaling indicates that unlike the general-purpose processors, specialized hardware has to be designed to utilize the maximum potential of a given hardware budget rather than to rely on extra hardware to gain performance. Based on such an approach, the specialized hardware can continue delivering gain for data- and compute-intensive dense and sparse problems.*

To efficiently accelerate a problem, two ingredients are necessary (i) a balance between the computation rate and memory bandwidth, and (ii) an appropriate execution model. Since the key idea behind the specialized hardware is to not waste the compute and memory budgets, a balance between bandwidth and compute rate is necessary. This is possible by enabling stream accesses to memory from one hand by using software techniques and accelerating the computation on the other hand by using hardware techniques so that they

(data transfer and computations) work at the same pace. The second ingredient (i.e., the execution model) is closely related to the fundamental philosophy behind the specialized hardware, which supposed to have specific functionality. As the variation in the applicability of a given accelerator would be the variations in data – not the variation in the main operation, a data-driven execution model is preferred.

We seek to implement the mentioned key components on a wide range of on-demand sparse problems that we categorize into the recommendation systems, the inference of neural networks, iterative solvers of partial differential equations, and graph algorithms. The nature of each domain creates a new form of the problem. As a result, while trying to be efficient in using memory bandwidth and computation, we propose to solve the following sub-problems:

- A category of sparse problems with low data reuse rate have can potentially benefit from state-of-the-art technologies such as near-data processing. However, the *lack of spatial locality* because of sparse accesses to memory prevent benefiting from such efficient technologies.
- In a group of sparse problems, the efficient utilization of memory bandwidth is often the key challenge mainly because of a bottleneck in computation. Such bottlenecks that prevent benefiting from high memory bandwidth often stem in *dependencies* in computation and slow decompression mechanisms.
- Some of the sparse problems have the potential of significant parallelism. However, the *distribution* of non-zero elements in their data structure prevents them to fully benefit an efficient concurrent compute engine. In this case, even though data can be streamed efficiently, the peak *throughput* cannot be achieved.
- In super sparse data structures, using compression formats is highly beneficial in terms of storage and data transfer. However, when it comes to computation, extra

overhead of decompression is required to find the original locations of non-zero values. *Decompression* is not necessarily fast, and can create a performance bottleneck.

Some observations on the sparse problems help us to address the challenges. For instance, while the third group has the potential of revealing high compute-per-data-movement ratios, and they *allow modifications in the sparse structure*, the second group exhibit low compute-per-data-movement ratio and has stricter non-zero patterns, but they offer the chance of *transforming the mathematical expression*. Besides, we see that sparsity is orthogonal to compute-per-data-movement ratios. Therefore, the first group of sparse problems has the potential of utilizing efficient hardware designed for *dense* computation. A state-of-the-art example of such hardware is systolic arrays. Our contribution to support the rhythmic synchronous flow of data from memory to systolic arrays is to modify the distributions of the non-zero values rather than minimizing overall flops or memory footprint.

To improve the bandwidth utilization of the second group of sparse problems with data-dependent computations, we propose to extract more parallelism by mathematically transforming the computation to equivalent forms that can be easily rearranged to gain performance. Base on such transformation, we propose a light-weight reconfigurable engine that rapidly switches between the parallel and sequential parts of the sparse problem. The compute engine works with our locally-dense storage format, analogous to blocked storage formats, but with ordered values. The proposed storage format does not benefit from a specific pattern (e.g., diagonal) in sparse data structures. Instead of the traditional encoding and decoding schemes of sparse formats based on the indices of non-zero values, we propose using indices for (i) configuring the data flow simultaneously with data streaming, and (ii) placement of outputs. As a result, we use the whole available memory bandwidth for transferring payload data. Besides the reconfigurable accelerator, we proposed a fast decompression mechanism that bridges the streamlines of memory to a dense concurrent compute engine and works with well-known storage formats, supported in Python libraries.

In summary, We seek to achieve close-to-ideal performance for sparse problems, by

making the following software/hardware contributions:

- Processing data while we gather them from random locations of memory neither were they reside nor where dense computations occur.
- Mathematically transforming sparse computation to capture more parallel patterns and running the outcome using a light-weight reconfigurable engine that enables high utilization of memory bandwidth for scientific and graph applications.
- Proposing structured pruning algorithms for CNNs that captures dependencies in data to satisfy the data reuse patterns provided by systolic arrays.
- Sustaining a balance between computing latency and data transfer rate by avoiding streaming the unnecessary *zero* elements, and implementing fast decompression.

CHAPTER 1

INTRODUCTION AND BACKGROUND

Since sparse problems with several patterns of indirect memory accesses fail to effectively run on modern high-performance computers, software optimizations and hardware accelerators have been proposed to improve locality in memory accesses and enable a high degree of parallelism. However, the following unexplored challenges still exist in sparse problems, addressing which are the objective of this research. The first challenge is the lack of spatial locality that limits utilizing state-of-the-art hardware techniques for efficiently executing large-scale sparse problems. The second challenge is the limited opportunity for parallelism because several data dependencies limit utilizing the high bandwidth of the memory. The third challenge, which often occurs for the sparse problems with high data-reuse patterns (e.g., the inference of neural networks) is that the distribution of non-zero elements does not automatically match the pattern of recurrence flow of data into dense computation hardware. Finally, the compression of sparse data, often implemented for reducing memory footprint, creates a performance bottleneck. In short, the performance of running sparse problems that are memory-bounded are limited by the preceding obstacles that prevent a continuous stream of memory access. Our key insight to address the aforementioned challenges is that based on the type of the problem, we can either modify the distribution of non-zero elements, transform the computations mathematically, or change their representations. By applying such techniques, while maintaining the nature of the problem, the execution adapts more effectively to given hardware resources. Therefore, this research introduces hardware/software techniques to enable stream access to memory for accelerating sparse recommendation systems, iterative solvers of partial differential equations (PDEs), deep neural networks (DNNs), and graph algorithms.

1.1 Sparse Problems

Sparse data and randomness in the nature of a problem are the two main sources of sparsity and irregular memory accesses, resulting in the major problem of costly data movement [1, 2, 3, 4, 5, 6, 7, 8]. The challenge of irregular memory accesses, first identified as a major challenge in the high-performance computing (HPC) community [9], is now an obstacle to a wider range of applications in which *randomness in the nature of a problem* contributes to creating irregular memory accesses, with applications in various domains, from robotics to recommendation systems, a growing application domain that captures sparse gathering. This category of sparse problems includes a data set that randomly gets accessed. For instance, in robotics, while a robot moves and observes objects, it needs to access data corresponding to those objects which are not necessarily co-located in memory [8]. Therefore, in this category, data is dense, but the *pattern of accesses to memory is sparse*. In the following, we first describe sparsity in recommendation systems, after which we explain sparsity in the other category of sparse problems (i.e., scientific computing, neural networks, and graph analytics), in which data is sparse.

To recommend content such as music, video, and products to users, recommendation systems are broadly used throughout industry [10, 11, 12]. The recommendation systems consist of (i) embedding tables, the sets of embedding vectors that contain users' data and features, followed by (ii) neural networks, including fully connected [13] and/or rectified-linear-unit [10] layers. To recommend content, first, the related embedding vectors are gathered from the embedding tables. Then, a simple reduction operation (e.g., element-wise summation, minimum, average) is applied to the gathered embedding vectors to derive a single vector, which is then sent to the neural networks for further processing. While embedding tables are dense, looking them up causes random memory accesses, which results in the sparse gathering. Therefore, embedding lookup is the *sparse* and the memory-bandwidth-hungry part of the recommendation system. Besides, the embedding lookup

and the reduction operations capture lower computation intensity and more cache misses compared to neural networks [12]. Such characteristics put recommendation systems in the memory-bound region of the roofline model of CPUs and far below the ceiling [14] because of memory bandwidth underutilization.

Recommendation systems that create a big portion of execution cycles at data centers, for instance, 65% of artificial-intelligence cycles in Facebooks data-center [12], are just one example of sparse problems. Sparse problems are everywhere. For instance, scientific computing and graph analytics that create more than 96% of todays supercomputer workloads are sparse, too. In scientific computing, solving partial differential equations (PDEs) is a key component, which is used for modeling physical phenomena in several domains from biology to chemical science. To model a physical phenomenon using PDEs and solving them using digital computers, the first step is to discretize it into a grid and represent them as a linear system of algebraic equations: $Ax = b$. Discretization, however, does not occupy all the points in a 3D grid. As a result, *the coefficient matrix A , used for representing the PDEs as a system of linear equations is sparse*. When such a system is too large to be solved by direct methods, they can be solved by iterative algorithms such as pre-conditioned conjugate gradient (PCG) methods. The execution time of the PCG algorithm is dominated by two kernels, SpMV, a parallelizable algorithm, and SymGS that includes several dependencies. By assuming a vector ($b_{1 \times m}$) and a matrix ($A_{m \times n}$) as operands, each elements of output vector $x_{1 \times n}$ is $x_j = \sum_{i=1}^k b[A^T_ind_i] \times A^T_val_{ij}$ for SpMV, and $x_j^t = \frac{1}{A_{jj}^T} - (b_j - \sum_{i=1}^{j-1} A_{ij}^T \times x_i^t - \sum_{i=j+1}^n A_{ij}^T \times x_i^{t-1})$ at each iteration (t) of SymGS.

In some other cases, such as in neural networks, although initially, data is not sparse, we make them sparse to eliminate storing unnecessary data and unnecessary computation. Neural networks are becoming more and more popular in many applications from image processing to financial problems. Once we train a neural network, several weights get close-to-zero values. As a result, a common practice is to prune those small values because they do not impact the end result. Therefore, *the 2D matrix representation of the neural*

network is sparse. A popular example of neural networks is convolutional neural networks (CNNs). The sparse inference of CNNs can be done by using well-studied sparse matrix multiplication approaches that are applicable to inference by converting convolutions to $O_{K \times WH} = \mathbf{W}_{K \times F^2 C} \times \mathbf{I}_{F^2 C \times WH}$, in which K filters of size $F \times F \times C$ are applied to $W \times H \times C$ images. Accelerating this general matrix-matrix multiplication (GEMM), the first operand of which is sparse, is one of the goals of this research.

Graph algorithms are the other sparse problems with several applications from social science to biology and physics. Since in a graph, not all the nodes are connected, *the adjacency matrix that represents the edges of a graph is sparse.* The sparse adjacency matrix is processed by algorithms such as breadth-first search (BFS), single-source shortest path (SSSP), and page rank (PR), all of which are similar to SpMV.

1.2 Contributions

The contributions of this thesis revolve around efficiently accelerating the execution of sparse problems that as mentioned earlier, are the main component in several crucial domains such as robotics, recommendation systems, machine learning, computer vision, graph analytics, and scientific computing that remarkably impact human life. For instance, modeling/simulating a vaccine or predicting an earthquake are examples of sparse scientific computing that can save lives if done accurately and promptly. In 2020, several supercomputers from Google Cloud, Amazon Web Service, Microsoft Azure, and IBM, running on over 136 thousand nodes containing five million processor cores and more than 50 thousand GPUs [15] for vaccine development is compelling evidence of the importance of sparse scientific computations. However, modern high-performance computers equipped with CPUs and/or GPUs are poorly suited to these sparse problems, utilizing a tiny fraction of their peak performance (e.g., 0.5% - 3% [16]). Such conventional architectures are mainly optimized to handle *complex computation* rather than the *complex memory accesses* that are essential for sparse problems. The contradiction between the abilities of the hardware and

the nature of the problem causes sparse problems to waste extra hardware budget (high power and dollar cost) for higher performance. The goal of this thesis is to propose effective solutions that utilize the maximum potential of a given hardware budget to accelerate sparse problems. To be impactful in achieving the goal, our research suggests that software and hardware must be co-optimized. To date, several software- and hardware-level optimizations have been proposed to accelerate sparse problems; however, since they optimize either the software or the hardware in isolation, they have not fully resolved the challenges of sparse problems. Table 1.1 summarizes the common challenges of sparse problems, examples of sparse applications that suffer from these challenges, as well as the contributions of this thesis to resolving these challenges along with the broader impact that this thesis would have, as explained in the following.

Table 1.1: The challenges and applications of sparse problems, our contributions and publications to resolve the challenges, and the broader impacts that this research would have.

Challenges	Irregular and inefficient memory accesses	Data dependencies	Slow decompression	Computation underutilization
Application	Recommendation systems	Scientific computing and graph analytics		Computer vision
Our Contributions	Proposed an intelligent tree near memory to reduce data while gathering them	Converted mathematical dependencies into gate-level dependencies	Co-optimized compression format and hardware to establish a balanced streaming from memory	Modified the distribution of non-zero data rather than minimizing their count
Our Publications	Fafnir [HPCA'21]	Alrescha [HPCA'20]	Ascella [DATE'20]	Lodestar [DAC'19] Eridanus [IEEE Micro'19]
Broader Impact	Facilitating e-commerce, e-learning, entertainment, tourism, and healthcare	Accelerating large-scale, critical, and super slow tasks such as vaccine development or timely predicting natural disaster such as earthquake and hurricane		Enabling accurate and fast manufacturing, financial services, healthcare, and agriculture.

Fafnir — Irregular random memory accesses are an obstacle in large-scale applications, such as recommendation systems, that not only are widely used in industry for e-commerce and entertainment but also have broader applications in domains such as e-learning and healthcare. To reduce the amount of data movement and thereby better utilize memory bandwidth, previous studies have proposed near-data processing (NDP) solutions for recommendation systems. The issue of prior proposals, however, is that they either minimize data movement effectively at the cost of limited memory parallelism or improve memory parallelism (up to a certain degree) but cannot successfully decrease data movement, as they rely on spatial locality (an optimistic but not realistic assumption) to utilize the NDP.

Besides, neither approaches propose a solution for *gathering* data from random memory addresses; rather they just offload operations to NDP. To deal with the mentioned challenges, in **chapter 3**, we propose an efficient near-memory intelligent reduction (Fafnir)[17] tree, the leaves of which are all the units in a memory system, and the nodes gradually apply reduction operations while data is gathered from *any* memory unit. Fafnir does not rely on spatial locality; therefore, it minimizes data movement by performing *entire* operations at NDP and fully benefits from parallel memory accesses. Fafnir also offers other advantages such as eliminating redundant memory accesses without using costly and less effective caching mechanisms and being applicable to other sparse problems such as scientific computation and graph analytics. We implement Fafnir on an FPGA and in 7 nm ASAP ASIC. Our evaluation shows that Fafnir executes recommendation systems $21.3\times$ more quickly than the state-of-the-art NDP proposal. Besides, the generic architecture of Fafnir allows the execution of the classic sparse problems using the same proposed 1.2 mm^2 chip up to $4.6\times$ more quickly than the state of the art.

Alrescha — The next unsolved challenge in sparse problems (e.g., in scientific computing) is data dependency, which limits utilizing the available memory bandwidth. To minimize the negative impact of data dependencies on performance, in **chapter 4**, we propose a lightweight reconfigurable sparse-computation accelerator (Alrescha) [18], the key insight of which is to convert the mathematical dependencies to gate-level dependencies to reduce the critical-path latency. As a result, even dependent instructions can be executed mostly in parallel. Based on this insight, Alrescha breaks down a sparse problem into a majority of parallel and a minority of small data-dependent instructions that can be executed quickly (in fewer cycles) in hardware. Besides the fast execution of small data-dependent parts, Alrescha smoothly implements the switching between two groups of operations, which is the other essential requirement in achieving better performance. To do so, Alrescha modifies the execution order of operations. To provide a platform with the aforementioned characteristics, Alrescha makes two main contributions. First, it implements a compute engine

with a fixed compute unit for the parallel parts and a lightweight reconfigurable engine for the execution of the data-dependent parts. Second, Alrescha benefits from a locally dense storage format, with the right order of non-zero values to yield the order of execution dictated by the hardware. The combination of the lightweight reconfigurable hardware and the storage format enables uninterrupted streaming data from memory. Our evaluations show that Alrescha executes sparse scientific computing, including solving partial differential equations, $15.6\times$ faster than GPUs. Moreover, compared to GPUs, Alrescha consumes $14\times$ less energy. Thanks to its dynamic and partial reconfigurability, Alrescha can accelerate problems such as graph analytics with wide applications in social sciences, linguistics, biology, and any problem that can be represented as a graph. Alrescha processes graphs $8\times$ faster than GPUs.

Lodestar & Eridanus — The next studied challenge of sparse problems is the underutilization of computation units, which concerns an isolated hardware optimization that is not effective without considering software. An example of such effort is accelerating the inference of convolutional neural networks (CNNs) using systolic arrays, the highly parallel arrays of multiplication and accumulation (MAC) units for performing matrix multiplication. However, the *dense* structure of systolic arrays contradicts the *sparse* nature of CNN inference and leads to underutilization of computation. To address this challenge, in **chapter 5**, we proposed creating locally-dense CNNs for efficient inference on systolic arrays (Lodestar) [5]. Lodestar is a structured pruning approach that produces CNNs, the non-zero values of which are clustered spatially into locally dense regions that can be compactly stored and efficiently streamed from memory. Lodestar consists of two key insights: (i) To capture the data reuse pattern in systolic arrays and enable data streaming, modifying the *distribution* of non-zero values is more influential than minimizing the number of operations or the memory footprint; and (ii) Examining the correlation among the filters rather than the individual filters increases the chances of creating a systolic-friendly model. To utilize Lodestar for streaming data from memory through systolic arrays, we proposed

efficiently running the inference of CNNs using systolic arrays (Eridanus) [19]. Eridanus handles the timing and indexing of streaming blocks of data. The evaluations show that Lodestar and Eridanus run CNN inference $8.4\times$ faster than the state of the art. Although by utilizing proposed techniques we can achieve the high *throughput* offered by the MAC-based systolic arrays, achieving low *latency* and scalability will still be key challenges. To resolve these challenges, we proposed multiplying matrices efficiently in a scalable systolic architecture [20], which compared to state-of-the-art systolic arrays, offers $1.83\times$ better single-batch inference latency by separating multipliers from the adders rather than combining them in a unified array of MACs.

Ascella — The other challenge of sparse problems concerns compression, a common software approach for storing and transferring sparse data. Although compression allows efficient data transfer, it can potentially create a computation bottleneck. This challenge is not only unresolved but also becomes more serious with the advent of domain-specific architectures (DSAs), as they intend to more aggressively improve performance. The performance implications of using compression along with DSAs have not been studied by prior work. To fill this gap of knowledge, in **chapter 6**, we introduce Ascella [21] that characterizes the performance implications of compression formats used in sparse workloads based on six key metrics, including memory bandwidth utilization, resource utilization, latency, throughput, power consumption, and balance ratio.

Ascella leads architects to knowingly choose the required sparse format and tailor their DSA for their target sparse applications, if necessary. For instance, this study helps accelerate sparse computation by enabling parallel stream accesses to memory, based on two main ingredients: (i) using a compression format that on one hand assures streaming only the non-zero values, and on the other hand, is easy to decompress using simple logic; and (ii) proposing a computation engine that follows the speed of memory streaming. To enable the latter, the central building block is optimizing the decompression

mechanism. Therefore, Ascella balances memory and computation latency while streaming blocks of data from memory into a DSA. More specifically, unlike in Figure 1.1a, where the memory and compute latency are not matched, in Figure 1.1b (Ascella), they are almost similar – the black and orange lines are close. Furthermore, Ascella reduces the maximum latency of streaming a block from 2200ns to 800ns as indicated by comparing the black line in Figure 1.1a and the lines in Figure 1.1b.

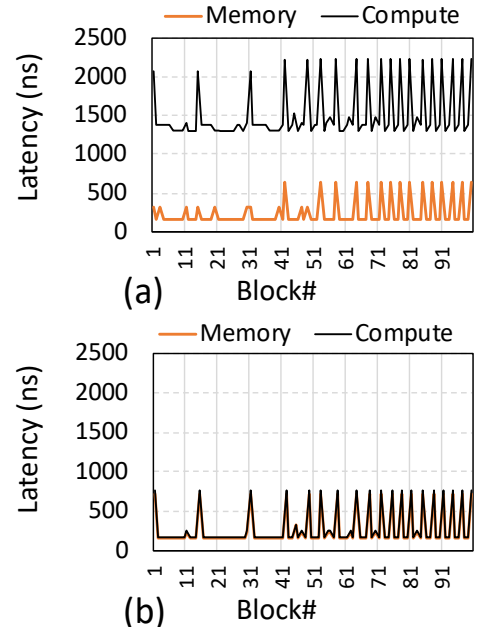


Figure 1.1: Memory and computation latency for streaming blocks for (a) prior work, and (b) Ascella. Closer memory and compute is better

1.3 Thesis Statement

Since sparse problems have specific patterns of memory accesses, they cannot run efficiently on the general-purpose processors. Thus, to keep gaining performance for sparse problems from a given hardware budget, hardware/software co-optimization is necessary.

CHAPTER 2

LITERATURE SURVEY AND THE GAP OF KNOWLEDGE

This chapter overviews the prior studies on sparse problem that are mostly related to the directions of this thesis. We mainly categorize them as specialized hardware and software techniques. First, this chapter summarizes the hardware solutions for sparse recommendation systems, neural networks, sparse matrix algebra, and scientific computing. Then, it outlines the software techniques for dealing with sparsity in neural networks as well as the common compression formats that are used in a wide range of sparse applications with various sparsity patterns.

2.1 Specialized Hardware for Sparse Acceleration

In this section, we overview the prior domain-specific hardware architectures that target various applications including recommendation systems, neural networks, sparse matrix algebra, and scientific computing.

2.1.1 Near-Data Processing for Recommendation Systems

Despite common efforts to reduce the embedding vector dimension [22] or the number of embedding vectors [14], the embedding tables occupy multiple gigabytes of memory. Such constraints necessitate the distribution of the embedding tables across multiple memory devices to satisfy memory capacity requirements [13]. Sparse gathering from random addresses scattered over a large memory system requires maximizing memory-bandwidth utilization. However, the *processor-centric* organization of CPUs and GPUs and the reused-optimized structure of their memory hierarchy, conspire against the efficient and fast sparse gathering. As a result, recommendation systems demand *data-centric* near-data processing (NDP) solutions to process data (i.e., the embedding vectors) where they reside.

NDP solutions have been explored to reduce costly data movement, a major problem of sparse problems [23, 24, 25, 6, 26, 27]. For instance, Gao et al. [23] have placed simple processing cores near hybrid memory cube (HMC) [28] and High Bandwidth Memory (HBM) [29] to propose an efficient and practical solution for data analytics including deep neural networks and graph processing. In another work, Gao et al. have proposed Tetris [26], a neural network accelerator that places an array of processing elements (PEs) close to HMC. To do this, it optimizes the size of the PE array to fit it near vaults of HMC. The programmability and scalability of NDP solutions for accelerating neural networks have been studied by Kim et al. [30] and Ahn et al. [27]. Other NDP studies have focused on accelerating graph and HPC problems, in which the majority of the operations (e.g., 80%) are sparse gathering [31]. To take advantage of 3D-DRAM-based NDP for graph processing, Nai et al. [6] have proposed GraphPIM, to offload instructions close to memory in a non-intrusive way without requiring programmers effort or ISA changes. Several other NDP solutions [2, 3, 4, 7] have proposed offloading computation to memory to reduce data movement and leverage NDP to accelerate data access and facilitate computations on sparse data structures [32, 33, 1]. Additionally, DIMMNet [34] has been proposed to accelerate gathering irregular memory accesses.

The aforementioned NDP solutions, however, are not very effective for embedding lookup of recommendation systems for several reasons. First, they decrease data movement by *rearrangement*, but do not perform *reduction* operations. Second, they are costly, as they copy a page to another in a scratchpad memory (e.g., [31]). Finally, they are not transparent to the software. Therefore, prior work has proposed specific NDP solutions, namely TensorDIMM [35], RecNMP [36], and Centaur [37] for recommendation systems. TensorDIMM [35] splits the embedding vectors across DIMMs to utilize rank-level parallelism for reading *individual* embedding vectors. Accordingly, it splits the reduction operations across the DIMMs. As a result, TensorDIMM successfully performs all reductions at DIMMs and minimizes data movement from memory to the cores by sending only out-

come vectors rather than all embedding vectors. However, the downside of TensorDIMM is that it does not sufficiently utilize memory parallelism because it uses column-major order, which fundamentally breaks the row-buffer locality in the DRAM system. As opposed to TensorDIMM [35], RecNMP [36] utilizes rank-level parallelism for reading *distinct* embedding vectors. Thus, the performance of RecNMP scales better as more ranks are added to the memory system.

Despite the advantages of NDP solutions for accelerating the inference of recommendation systems, they left unsolved a few key challenges of *sparse gathering*. In the following, we elaborate on different challenges, solving some of which have been the target of a prior study that in turn causes the other challenges. Data movement has been the main concern of sparse gathering. For instance, as Figure 2.1a shows, to apply two non-compute-intensive reduction operations (i.e., $v1 + v2 + v5 + v6$ and $v1 + v3 + v4 + v5$) on six embedding vectors (i.e., $v1, v2, v3, v4, v5$, and $v6$), the vectors must be transferred to cores. For example, vector 5 ($v5$) is required by two queries; thus, it is transferred twice. In general, to perform n queries of size q on vectors including v elements, $n \times q \times v$ elements must be transferred from the memory system to cores. To solve this challenge, TensorDIMM [35] performs reductions in the DIMMs and transfers only the results to the cores (Figure 2.1b). As a result, instead of transferring all q vectors in a query, it sends only one vector, hence reducing the amount of data movement q times to $n \times v$. TensorDIMM splits the embedding vectors and distributes them over DIMMs and creates $1/q$ of each output vector at a DIMM. Then, the cores just concatenate the partitions of the outputs.

Even though TensorDIMM effectively reduces data movement, it is not as effective in sufficiently utilizing row-buffer hits because it uses column-major order, which fundamentally breaks the row-buffer locality in the DRAM system. More specifically, while TensorDIMM can utilize rank-level parallelism to read the elements of individual embedding vectors, split over different DIMMs, it must access random rows to read *distinct* embedding vectors in a query (e.g. vectors of query 1 including $v1, v2, v5$, and $v6$). Accordingly, only

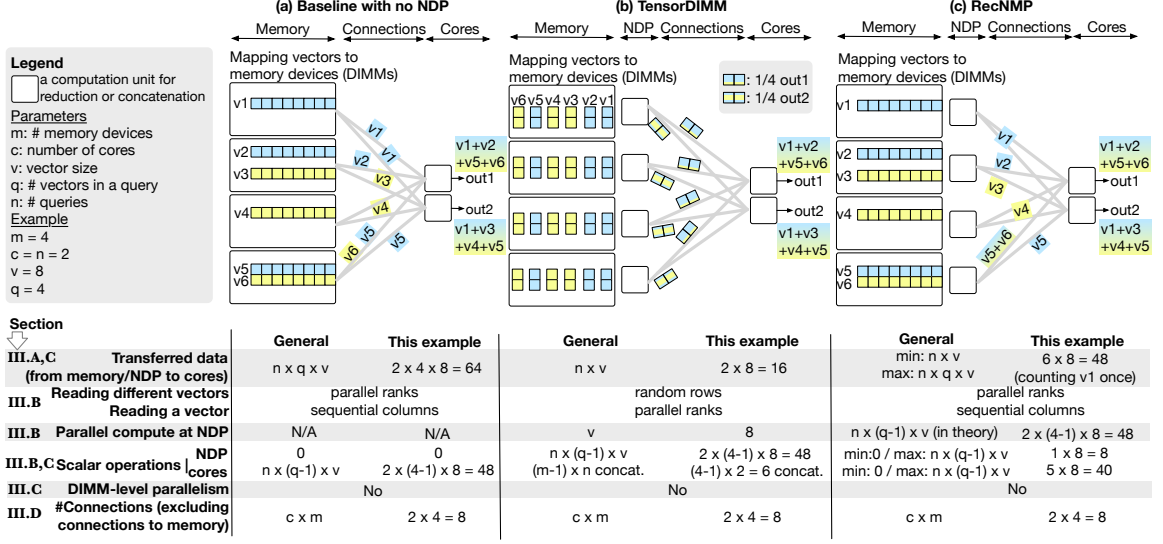


Figure 2.1: Comparing NDP-based solutions for embedding lookup: (a) Baseline with no NDP, (b) TensorDIMM [35], and (c) RecNMP [36].

v scalar operations can be performed in *parallel* at NDP. Although TensorDIMM performs all $n \times (q - 1) \times v$ operations at NDP, only v of them are processed in *parallel*, while the rest can be *pipelined*. For instance, for query 1, all DIMMs do the following subsequently: read their own part of v_1 from a row (but not necessarily reading the entire row buffer), then read v_2 from another row, do a partial sum of size v/m (v_1+v_2), simultaneously access another row to read v_5 , add it to the partial sum while reading v_6 from another row. This approach particularly disturbs achieving low latency.

Splitting embedding vectors across more ranks causes poor utilization of row-buffers (i.e., we must open a row, but read a *smaller* fraction of it). To improve parallel computation at NDP, RecNMP [36] distributes embedding vectors across the ranks, as shown in Figure 2.1c. In this approach, reading *distinct* embedding vectors utilizes rank-level parallelism, while elements of each vector are read from sequential columns. As a result, RecNMP can more effectively increase rank-level parallelism by adding more ranks to the system. However, the downside of this approach is that even though in theory entire operations for all queries (i.e., $n \times (q - 1) \times v$) can be performed in parallel at NDP, RecNMP might not achieve it because of imperfect spatial locality.

Although RecNMP utilizes rank-level parallelism in reading distinct embedding vec-

tors, it does not guarantee to process them *all* at NDP, mainly because it does not provide DIMM-level parallelism. In many real-world applications, embedding vectors of a query are scattered over many random DIMMs, where DIMM-level parallelism (i.e., channel-level reduction) is essential. For instance, based on the birthday paradox, the probability of having a query with indices on the same channel is only up to 25% in a four-channel system. Consequently, as in many cases, the raw data needs to be transferred to the cores, memory bandwidth may not be fully utilized. Thus, even though under perfect circumstances, maximum $n \times (q - 1) \times v$ operations can be done at NDP, in the worst case, all of them might need to be done at the cores. For instance, in Figure 2.1c, only two embedding vectors (i.e., v5 and v6) are reduced at NDP, and others are forwarded to the cores. Relying on spatial locality has two other consequences. First, increasing batch size does not necessarily result in more utilization of parallel computation at NDP, hence achieving higher throughput. Second, while in the perfect scenario only n output vectors (i.e., $n \times v$ elements) are transferred from the memory to the cores, in the worst case, all $n \times q \times v$ elements must be transferred. Therefore, reducing data movement is also not guaranteed.

The other challenge of implementing embedding lookup is the overhead of connections. As the embedding tables are often large, they necessitate model parallelism (i.e., splitting and distributing tables across memory devices), as shown in Figure 2.1a. On the other hand, the neural network layers of the embedding systems are small enough to utilize data parallelism (i.e., mapping copies of a neural network on different computing devices). The combination of model parallelism for embedding tables and data parallelism for neural networks in recommendation systems requires costly all-to-all connections [13] (e.g., Figure 2.1a) between the memory devices and computing devices (e.g., CPU or GPU cores) so that embedding vectors can be gathered from any memory device and be forwarded to any computing device. The previous studies have not proposed any solutions to reduce the number of connections. Therefore, similar to the baseline, they all require $c \times m$ connections to implement all-to-all communication, which is not only costly but also limits the

scalability. To accelerate sparse gathering and prevent the communication from becoming a bottleneck, Centaur [37] uses high-bandwidth communication links and then applies the reduction operations in a separate unit. Thus, unlike TensorDIMM, Centaur does not reduce data movement but instead transfers data more quickly.

The last challenge of implementing NDP solutions for embedding lookup is eliminating extra memory accesses. Observations suggest that a batch of queries has common embedding vectors. Thus, not all the memory accesses corresponding to every single embedding vector are necessary. RecNMP [36] proposes using caches at NDP. Caching, however, is not the most effective solution, as no more than a 50% hit rate can be achieved [36]. Furthermore, even achieving such a hit rate requires a 128 KB cache that adds extra hardware overhead (e.g., 38% area [36]). Besides, the cache accesses can potentially cause a performance bottleneck.

2.1.2 Hardware Accelerators for Neural Networks

Various types of hardware accelerators have been proposed to efficiently execute neural networks. In the following, we summarize five categories of them.

Systolic-based accelerators: Since 1979 [38] various architectures have been introduced for systolic architectures [39, 40]. More recently, the advantages of artificial intelligence and the need for massive parallel matrix multiplication have motivated academia and industry to rethink the systolic arrays [41, 42, 43, 44, 45, 46, 47, 48, 49]. Systolic arrays for matrix multiplication have also been implemented by industry in large data-center scales such as Google’s tensor processing unit (TPU) [49]. Regardless of the different implementations, the systolic-based matrix multipliers used in prior studies can be categorized as non-stationary and stationary, based on the way the operands of the matrix multiplication are being handled during execution. In the following, we explore both categories.

The processing elements (PEs) of non-stationary systolic array (NSA) systolic architecture (e.g., [50, 39, 40]) are multiply-and-accumulate (MAC) units. As its name indicates,

none of the inputs stay in the PEs during the execution, and they pass through the PEs in two different directions. Upon the arrival of new inputs, each PE multiplies the two inputs coming from its neighbors and adds them to the prior accumulated results. At the end of the execution, each PE contains one element of the output matrix. To guarantee the correctness of computations, the inputs must arrive at each PE at the right time. To do this, the two inputs are inserted into the array. Since both inputs are non-stationary, the multiplication starts as soon as the first elements of the inputs arrive at a PE. Therefore, no additional time needs to be spent on loading. To finish the multiplication, all elements of both inputs must pass through the PEs completely. Once the multiplication is done, the outputs generated in the PEs must be sent out.

A more popular type of systolic array for matrix multiplication is the TPU-style stationary systolic array (TSSA), which is the architecture of the systolic array in TPU [49]. TSSA is also called weight stationary [51] or static systolic arrays [52] and has been implemented for neural networks. Similar to NSAs, the PEs of a TSSA are MAC units. However, unlike NSAs, the PEs in TSSA keep one of the inputs in their registers and instead pass through their outputs. As a result, before starting the multiplications, one of the inputs must be loaded to the registers of each PE. Similar to the NSA, all elements of the output must be carried out even though they are being created and passed through the PEs.

Accelerators for inference: Several NDP-based hardware solutions have been proposed for the inference of neural networks that include but are not limited to [23][24][25][6][53][54][26] for relaxing data movement costs. Among them, Tetris [26], which optimizes the size of PE array to place it near vaults of HMC, and Neurocube [30], which is a programable HMC-based neuromorphic architecture are the most related NDP-based neural-network accelerators. Besides the NDP-based hardware, several other studies have also proposed solutions for inference of CNNs [55][56][44][47][48][57], which often use an array of compute units. Similar to Tetris[26], the architectures used in Eyeriss[55] and DianNao[56], devote a huge portion of each PE to SRAM memories. Therefore, they limited their PE array size

to 8×8 up to 16×16 . Even Tetris[26], which optimizes the PE array size, has 512 Byte register file per PE. As this paper showed, using a systolic array near memory that, for instance, uses only four bytes of register per compute unit, is more effective in allocating a limited hardware budget to compute logic rather than SRAM memory.

Accelerators for training: Scaleddeep [58] and Neurostream [59] are two recent studies that accelerate CNN training. Scaleddeep uses a heterogenous tile of chips to implement convolutional and fully-connected layers, and Neurostream uses a general-purpose clustered many-core platform. More specialized hardware for training neural networks is TPUv2 [57]. A TPUv2 board provides a peak of 180TFLOPS by employing four TPUv2 chips, each including two cores, connected to an 8GB HBM package at 300 GB/s. A relatively equivalent configuration of Mahasim (in terms of the size of the memory) integrates each 1GB stack of HMB with two compute units, each with a systolic array of size 8×256 (for the baseline), which delivers 524.288 TFLOPs/S from 2048 GB/s. For dense problems, this represents an approximately 2.91-factor improvement in peak throughput. This is due in part to the increased memory bandwidth to compute inside each package and greater compute density per GigaBytes of memory (45 TFLOPs/S vs. 131 TFLOPs/S per 8GB).

Accelerators for RNN/LSTMs: efficient speech-recognition engine (ESE) [60] improves the utilization of the a PE array by balancing the load across them. To do so, ESE employs a hardware unit that allows the faster PEs to fetch new elements and work on them, instead of waiting for slower PEs. A few other studies [61][62] have also proposed hardware accelerators for RNN inference and training based on arrays of MACs, and implemented them on FPGAs without specific focus on the scalability feature.

2.1.3 Hardware Accelerators for Sparse Matrix Algebra and Scientific Computing

The ineffectiveness of CPUs and GPUs, along with approaching the end of Moore’s law, has motivated the migration to specialized hardware for a wide range of sparse problems. For instance, hardware accelerators have targeted sparse matrix-matrix multiplication [5,

63, 64, 19], matrix-vector multiplication [65, 66, 67, 68], or both [69, 70, 71], which are the main sparse kernels in many sparse problems. A state-of-the-art SpMV accelerator, OuterSPACE [69], employs an outer-product algorithm to minimize the redundant accesses to non-zero values of the sparse matrix. Despite the speedup of OuterSPACE over the traditional SpMV by increasing the data reuse rate and reducing memory accesses, it produces random access to a local cache. To efficiently utilize memory bandwidth, Hegde et al. proposed Extensor [70], a novel fetching mechanism that avoids the memory latency overhead associated with sparse kernels. Song et al. proposed GraphR [72], a graph accelerator, and Feinberg et al. proposed a scientific-problem accelerator [73], both of which process blocks of non-zero values instead of individual ones. Besides, Huang et al. have proposed analog [74] and hybrid (analog-digital) [75] accelerator for solving PDEs. Moreover, many processing-in-memory studies [76, 2, 4, 6, 1] proposed offloading computation to memory to reduce the computation energy of sparse problems. The prior specialized hardware designs often have not focused on resolving the challenge of data-dependent computations in sparse problems that prevent benefiting from the available memory bandwidth.

Table 2.1 compares the most relevant hardware approaches and techniques for accelerating scientific computing and graph analytics. Several factors such as the compression format, the ability to resolve dependencies in computation (i.e., resolving limited parallelism), the range of applicability of a hardware/software technique, and reconfigurability impacts stream accesses to memory and are important in ideally accelerate sparse problems so that they meet their requirement. Transferring meta-data is defined not only by the storage format but also by the scheme of implementation. The storage format and the scheme for implementing it, together with impact the bandwidth utilization. The other insight obtained from comparing the aforementioned hardware accelerators and software-optimization techniques for sparse problems it that they often focus on a specific domain of application and take advantage of the specific pattern in computations to improve performance. However, flexibility in the range of target applications is an important feature for a

hardware accelerator. Such a *flexibility is important* not just for creating more generic accelerators; but, *for accelerating all the different kernels in a program* to effectively improve the overall performance.

Table 2.1: State-of-the-art approaches for accelerating graph and scientific problems

		GraphR [72]	OuterSPACE [69]	Memristive-Based Accelerator [73]	Row Reordering Matrix Coloring [77]
Application Domain		Graph	Graph (Only SpMV)	PDE Solver	PDE Solver
Hardware	Multi-Kernel Support	No	No	No	No
	BW Utilization	Low	Moderate	Low	Moderate
	NOT Transferring Meta-data	No	No	No	No
	Processing Type	ReRAM Crossbar	PEs Connected in a High-Speed Crossbar	Heterogeneous Memristive Crossbar	GPU Instruction
	Cache Optimizations For Frequently-Used Vectors	N/A	No	N/A	No
	Reconfigurability	No	Only for Cache Hierarchy	No	N/A
Techniques	Storage Format	4×4 COO	CSR	multi-size blocks (64×64, 128×128, 256×256, 512×512)	ELL
	Resolving Limited Parallelism	N/A	N/A	No	Instruction-Level Limited by NNZ patterns

2.2 Software Techniques for Dealing with Sparsity

To date, many software-level optimizations for CPUs [78, 79, 80], GPUs, [81, 82, 83, 84, 77], and CPU-GPU systems [85] have been proposed for accelerating sparse problems. For instance, to reduce memory-access latency, Graphicionado [86], a graph-processing accelerator, substitutes accesses to the memory hierarchy with sequential accesses to scratchpad memory. Besides, techniques such as matrix coloring [77] and blocking [87] have been proposed to extract more parallelism for accelerating sparse problems. Additionally, to relax sparse gathering in graph applications, batching the accesses to the output vector and restricting them to a localized region of memory [88] has been proposed. In the following, we focus on pruning the neural networks and compression formats that are the techniques most related to this thesis.

2.2.1 Pruning the Neural Networks








The dense computation structure in systolic arrays is contradictory with the sparse data structure of CNN, even though their high data-reuse rates match. To make the two more

compatible, the column-combining approach [51] prunes all weights on conflicting rows for a selected set of columns except the one with the largest magnitude. The other pruning techniques, however, have not been proposed for efficiently using the systolic arrays for CNN inference. For instance, structured sparsity learning (SSL) [89] formulates shape-wise pruning as well as pruning in the granularity of kernel, filter, channel, and layer, and reported more than $3\times$ speedups on CPUs and GPUs while sustaining accuracy. Other efforts [90, 91, 92, 93] have studied filter-wise pruning by applying various implementation techniques. Examples of filter-wise pruning are pruning the model based on the global rescaling of a criterion (e.g., the mean, standard deviation) for all layers [93], or selecting close-to-zero weights based on the smallest sum of absolute values [90]. In another work, Scalpel [91] implements filter-wise pruning for hardware with high parallelism (e.g., GPU), weight grouping for single instruction, multiple data (SIMD), and a combination of both for hardware with moderate parallelism (e.g., CPUs).

Table 2.2 compares using the prior pruning methods when targeting systolic arrays. While element-wise pruning does not guarantee *any* spatial locality, vector-, and kernel-wise pruning do not capture the spatial locality *required by systolic arrays*, which results in underutilization of a systolic array are active during inference. The storage overhead and indexing complexity that are defined by the granularity of pruning is low (low is better) for kernel-, filter-, and channel-wise pruning. On the other hand, the opportunity of *concurrency* and *data reuse* is defined by the shape of the pruning granularity. For instance, row-wise proximity offers higher concurrency while column-wise proximity captures more data-reuse patterns in systolic arrays. Although kernel-, filter-, and channel-wise pruning methods offer a high level of concurrency, they might not match the concurrency required by the algorithm. Finally, since none of the pruning methods offer a granularity width that a systolic array prefers, they all require some sort of buffering/caching mechanisms *to enable efficient streaming from memory*. In addition to that hardware complexity, when the granularity of pruning is small, ensuring correct timing to supply data from memory to the

compute units of systolic arrays requires complex indexing hardware. According to how the compiler orders the elements of weight matrices, the listed granularities in Table 2.2 could be inferred as the same implementations, but they still do not capture the specific structured-pruning, required by systolic arrays.

Table 2.2: The impact of pruning granularity on inference using systolic arrays.

Convolution Layer Weights	Granularity	Element-wise	Shape-wise	Vector-wise	Kernel-wise	Filter-wise	Channel-wise	Our Proposal
		 $\mathbf{W}(f_1^{th}, f_2^{th}, c^{th}, k^{th})$	 $\mathbf{W}(f_1^{th}, f_2^{th}, c^{th}, :)$	 $\mathbf{W}(f_1^{th}, :, c^{th}, k^{th})$	 $\mathbf{W}(:, :, c^{th}, k^{th})$	 $\mathbf{W}(:, :, :, k^{th})$	 $\mathbf{W}(:, :, c^{th}, :)$	 $\mathbf{W}(m_1..m_2, k_1..k_2)$
Systolic Architecture	Pruning Opportunity	High	Low	Low	Low	Low	Low	Medium
	% Active Units	Low	Medium	Low	Low	Medium	Medium	High
	Storage Overhead	High	Medium	Medium	Low	Low	Low	Low
	Concurrency*	Low	Low	Low/Medium	Low/Medium	Medium	Medium	High
	Data Reuse	Low	High	Low	Low	Low	High	High
	No Buffering/Caching	No	No	No	No	No	No	Yes
	Indexing Complexity	High	Medium	Medium	Low	Low	Low	Low

* Whether or not the level of concurrency in the resulted pruned weight could be matched with that of the underlying systolic array.

2.2.2 Compressing Sparse Matrices

Since the primary issue with sparse matrices is storing the enormous amount of non-necessary zero elements, several compression formats have been proposed to efficiently store sparse matrices. While some formats target generality, others are tailored for particular patterns of sparseness (e.g., diagonal matrices) to be more effective in saving them with minimum storage overhead. Such optimizations for sparse problems mainly focus only on the *storage* overhead in isolation without involving other essential performance metrics such as latency, throughput, and power efficiency. That said, a slower-than-data-transfer decompression can even surpass the overhead of processing all zero entries in the original dense matrix format; thus, sparse formats may not necessarily guarantee fast execution. This occurs because common sparse formats are often tailored to *the distribution of data*, not *the underlying mechanism of computation*.

The challenges associated with using sparse formats not only are not resolved with the advent of domain-specific architectures (DSAs) for sparse problems but also gain more

importance. DSAs seem to soon become the main platform of sparse computations by approaching the end of Moore’s law, proven by the tremendous number of recent studies [63, 64, 19, 66, 67, 68, 18, 69, 70, 71, 94], to more efficiently accelerate the execution of sparse problems. Prior studies [68, 95, 96, 97, 98, 60, 99] have also demonstrated the importance of fast compression/decompression in on-demand applications such as in the inference of neural networks. Regardless of this ongoing research, no study has shed light on the performance implications of using a variety of sparse formats. Particularly, even though prior work has studied the performance implications of software implementations of sparse formats [100, 101, 102, 103, 104, 105, 106], the *hardware* implementation of these formats on FPGAs has not been extensively characterized.

The compression format proposed by prior work lies on a spectrum from the dense format, which preserves the locations of non-zero values in the physical address of memory, to a more aggressive compressed format such as COO and CSR. In the following, we briefly introduce six frequently-used formats.

Compressed Sparse Row/Column (CSR/CSC): The CSR/CSC sparse format sequentially stores values in row/column order in a `values` array while similarly storing their column-index/row-index in a `indices` array. Another array, `offsets`, stores index pointers or range to create rows/columns. To do so, the adjacent pair of this array [`start:stop`] represents a slice from the two first arrays. Figure 2.2b shows an example of CSR. For an $n \times n$ matrix, the length of `offsets` is n (often $n + 1$, but the first element can store absolute value to reduce the size) and the *maximum*[†] length of `values` and `indices` is n^2 .

Block CSR/CSC (BCSR/BCSC): The block(-wise) compressed sparse row/column (BCSR/BCSC) [87] sparse format is similar to CSR/CSC, but arrays are stored based on the same-shaped blocks (sub-matrices) rather than on the original matrix. This allows block-wise formats to better deal with large matrices. Figure 2.2c illustrates an example of BCSR for block sizes of 4×4 , the block size we choose in all our experiments as well. For an

[†]Note that these worst-case scenarios are used for on-chip memory allocation. The *storage* overhead is still defined by the number of non-zeros.

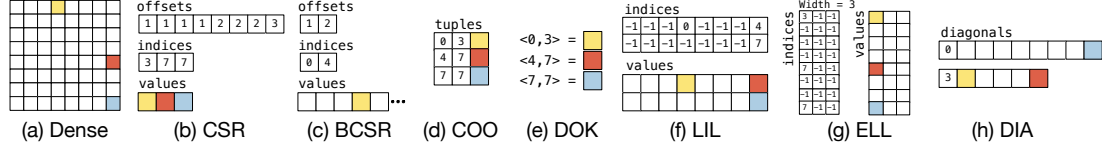


Figure 2.2: *Sparse formats*: (a) The original sparse matrix in **dense** format, (b) **CSR** including offsets to indicate the number of non-zero entries per row, indices, indicating column indices of non-zero entries, and the values itself. CSC follows the same rule as CSR; (c) **BCSR** including offsets to indicate the number of non-zero 4×4 blocks per row, indices, indicating the index of the first column of non-zero blocks, and the values, indicating the flatten values of non-zero blocks; (d) **COO** including a series of (row, column, value) tuples for non-zero values; (e) **DOK**, which is similar to COO; (f) **LIL**, which pushes all the non-zero entries to top and saves the row indices; (g) **ELL**, which is similar to LIL but pushes the non-zero entries to left and also uses a padding; and (h) **DIA**, which saves the non-zero diagonals by adding the diagonal numbers as a header to each diagonal.

$n \times n$ matrix and $b \times b$ blocks, the length of `offsets` is n/b and the *maximum* length of `values` and `indices` are n^2 and $(n/b)^2$.

Coordinate (COO) & Dictionary of Keys (DOK): The COO sparse format simply stores a series of `tuples`, including the row index, column index, and value for each of the non-zero entries. For an $n \times n$ matrix, the *maximum* length of `tuples` is $3n^2$. The DOK format is similar to the COO format except that it stores coordinate-data information as key-value pairs. DOK uses hash tables to store a value with the key of (row index, column index). Figure 2.2d and e depict an example of COO and DOK, respectively.

List of List or Linked list (LIL): The LIL sparse format stores one list of non-zero elements per row/column. Each element in the lists stores the column/row indices of that row/column, `indices`, and their value, `values`. Figure 2.2f presents an example LIL, which compresses the rows and preserves the columns (this is our assumption for LIL). For an $n \times n$ matrix, the *maximum* length of `values` and `indices` is n , with n list in total.

Ellpack (ELL) & Sliced ELL (SELL): In the ELL [107] format, non-zero elements are extracted similarly to those of the LIL format, with their column indices and their values. However, they are stored in column-major format with the addition of explicit zero paddings to hold the data for the longest row. This format is ideal for SIMD units since the widths of all `values` and `indices` are the same. A sliced ELL (SELL) sparse format first

slices the dense matrix row-wise in chunks, and then applies ELL on each chunk. Hence, it reduces the overhead of zero paddings for larger matrices. Figure 2.2g shows an example of ELL with a padding width of three. We set this width to six. For an $n \times n$ matrix, the *maximum* length of `values` and `indices` is n (longest possible row). The width in ELL is n , and that in SELL varies based on the pattern of data. Variants of ELL formats such as ELL+COO, Jagged Diagonal Storage (JDS) [108], and SELL-C- σ [109] are also popular. ELL+COO mixes ELL and COO formats to reduce the width of long rows. The JDS format sorts the rows in ELL from longest to shortest (for vector machines). SELL-C- σ is a variant of JDS that only sorts rows within a window of σ .

Diagonal (DIA): The DIA [110] sparse format operates by specifying a diagonal number (0 for the main diagonal, negative/positive for diagonals which start on a lower/higher row/column) followed by the values that fall on the diagonal, `diagonals`. Figure 2.2h illustrates an example of DIA. For an $n \times n$ matrix, the *maximum* number of non-zero diagonals is $2n - 1$ and the *maximum* length of a diagonal is $n + 1$ (the additional element contains the diagonal number).

2.3 Unsolved Challenges

Despite the advantages of mentioned prior work for improving the performance of sparse problems, we believe that the following aspects still need to be studied: (i) The cost of data movement and ineffectiveness of NDP solutions for sparse gathering (e.g., embedding lookup in recommendation systems) demands a mechanism to process data neither where data resides nor where dense computation occurs; (ii) Dependencies in computations on sparse data are a crucial source of bandwidth underutilization and require more software and hardware optimizations; (iii) Decompressing sparse data even those represented in less aggressive storage formats, create a bottleneck in data streaming and requires fast decoding mechanisms; and (iv) Systolic arrays, the efficient compute engine for accelerating the inference of CNNs, necessitate pruned models that capture their interacting data flows;

This section sheds some light on the mentioned requirements and clarifies why the absence of them are the key challenges in accelerating sparse problems.

2.3.1 The Underutilization of Memory Bandwidth

Lack of spatial locality: As the comparison made in Figure 2.1 showed, on the one hand, we do not want to process embedding vectors in the processing cores where dense computation occurs because it causes a huge amount of data movement. On the other hand, we cannot process randomly scattered sparse data where they reside because they are not co-located in memory. In other words, either way would lead to a large amount of data movement. The negative impact of not reducing data movement is more pronounced at larger scales. The diagram in Figure 2.3 illustrates the time to gather random data from embedding tables when we add more memory units or ranks to the system of the most recent prior NDP solution for recommendation systems, RecNMP [36]. Numbers are normalized to a system of only one rank. Ideally, we want this time to decrease linearly as shown in the red line, when the memory system scales up. However, as the diagram shows, the performance of an NDP solution can get far away from the ideal linear speedup as the memory system size increase if such an NDP solution does not guarantee to apply reduction operations on embedding vector at near memory.

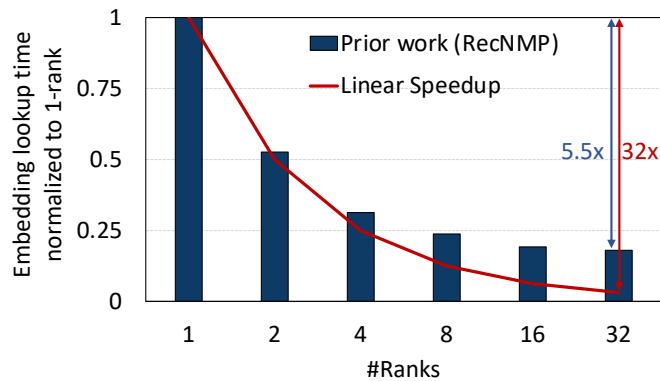


Figure 2.3: The time to perform embedding lookup in recommendation systems when the size of memory system grows. Numbers are normalized to a one-rank system.

Dependency in computations: Solving PDEs using the SymGS method can be written as

an extremely simplified expression that still sustains the dependencies:

$$x_i = \sum_{j=0}^{columns} A_{ij}^T \times x_j. \quad (2.1)$$

The equivalent code for implementing and processing Equation 2.1 in a computer would include a nested loop: the outer loop over the rows of A^T (i.e., $i=0$ to rows), and the inner loop over the columns of A^T (i.e., $j=0$ to columns). While the iterations of the inner loop can be parallelized, the iterations of the outer loop cannot, because of the data dependencies between them. Figure 2.4 demonstrates the dependencies between the iterations of the outer loop. The main cause of such dependencies is that at each iteration of the outer loop, we read the entire vector x (Figure 2.4, left) and then, we update one element of x (Figure 2.4, right). Therefore, before reading x , we must wait until it is updated.

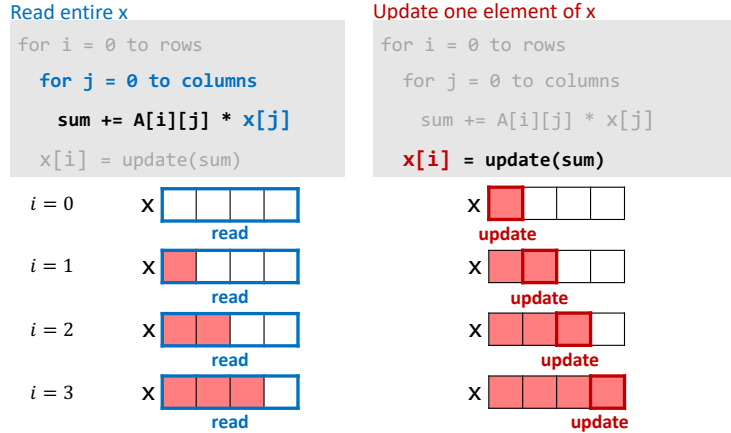


Figure 2.4: Dependencies in SymGS: Each iteration of the outer loop reads the entire vector x (left) and updates one element of x (right). The iterations of the outer loop are dependent.

As a result of such dependencies, executing this nested loop cannot benefit from the parallelism provided by GPU by employing common techniques such as loop unrolling. Figure 2.5 shows an example, in which we unroll the outer loop three times, and assume that a GPU has nine parallel processing units, three of which are used to process the inner loop. Since the iterations are dependent, three steps are required for processing the three unrolled iterations, at each of which, only is one-third of the GPU utilized.

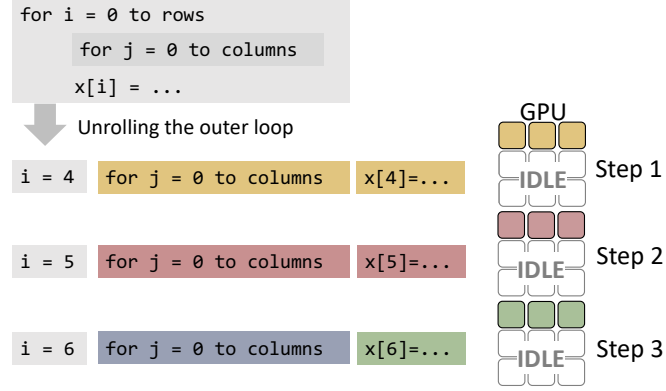


Figure 2.5: Limited parallelism: Unrolling the iterations of the outer loop (left) and mapping the parallelizable iterations of the inner loop to processing units of a GPU (right).

A deeper look at the pattern of dependencies in Figure 2.4 and the ineffectiveness of a parallel processor to execute the nested loop quickly, suggests that *not all the operations at each iteration of the outer loop read the newly updated elements by the previous iterations* (Figure 2.6a). Based on this observation, we add *blocking* as another optimization on top of the unrolling. As Figure 2.6a shows, a block of operations including the iterations of the inner loop (e.g., $j=4, 5, 6$) that depend on the outcome of the previous iterations of the outer loop (e.g., $i=4, 5, 6$), can be excluded from the other operations (i.e., the green part in Figure 2.6a). The width of the blocks determines the depth of the unrolling as well as the iterations of *inner* loop that are excluded.

Even though based on the key observation, more parallel operations can be extracted from the target nested loop, the effort cannot help improve the performance on GPUs. Figure 2.6b clarifies this by mapping parallelizable operations into the processing units of the GPU. As illustrated, even though running the green part in parallel increases the level of parallelism at step 1, the rest of the operations still need to wait for the previous steps, which take three additional steps. Therefore, implementing the unrolling and blocking can even worsen the execution time (i.e., four steps versus three). This paper seeks to address this challenge and enable benefiting from the key observation and the resulting software optimizations (i.e., unrolling and blocking) with the aid of hardware.

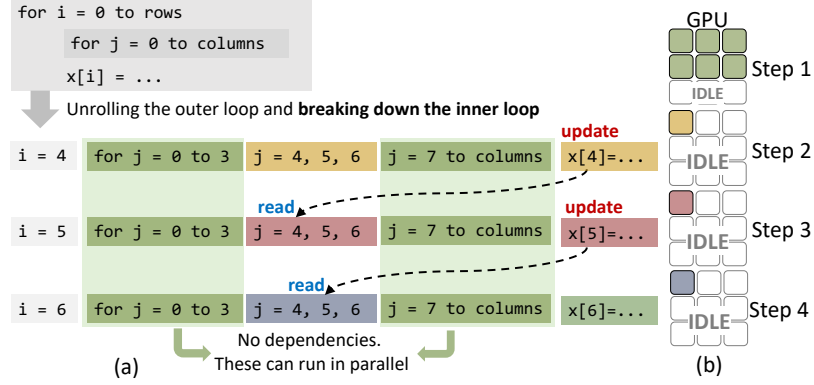


Figure 2.6: (a) Key observation: the iterations of the outer loop are just *partially* dependent. In fact, only a few iterations of the inner loop read the newly updated elements. Therefore, we can break down the iterations of the inner loop across a few unrolled iterations of the outer loop, into data-dependent part ($j = 4, 5, 6$), and parallelizable part (green parts); (b) Key challenge: ineffectiveness of blocking technique on GPUs. More parallelism at step 1, but dependencies still create the bottleneck through steps 2, 3, and 4.

Overhead of decompression: Figure 2.7 uses an example of SpMV to clarify the overhead of decompression for two popular storage formats CSR and BCSR formats. As mentioned earlier, CSR uses row indices to indicate the number of elements of each row. Therefore, for decompressing a non-zero row, we need to first read one element of row indices, and based on that, we can read as many column indices/values as required. As a result, the issues with CSR that makes it compute bounded is that (i) *an overhead of one access to the row indices and one computation is always required for all rows*; and, (ii) *accesses to the column indices and values are sequential*, because we do not know in advance which elements of column indices and values are going to be accessed in parallel.

The decompression steps of BCSR follow similar steps, whereas instead of individual non-zero elements, we decompress the non-zero sub-blocks. In the example of Figure 2.7, similar to CSR, one access to the buffers is required per each row of sub-blocks to initiate the next accesses for obtaining column indices and values. The advantage of BCSR over CSR is that the sub-blocks can be distributed over blocks of BRAM and be accessed in parallel. However, its downside is the overhead of transferring zero elements. In summary, the latency to process a $L \times W$ matrix partitioned into $l \times w$ sub-blocks is defined by

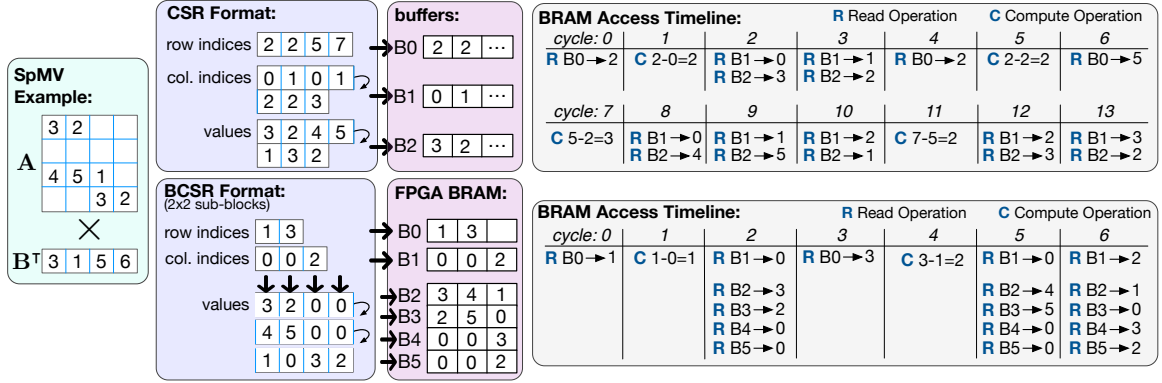


Figure 2.7: The time steps required to read a sparse matrix compressed in CSR and BCSR formats, for creating the non-zero rows. For simplicity, this example (and the next example in Figure 6.1) assume only one read per cycle from a buffer.

the overhead of accessing the buffer once per a rows-of-sub-blocks, and the latency of decompressing the non-zero rows-of-sub-blocks.

2.3.2 The Underutilization of Dense Computation

When using systolic arrays, the storage formats such as the CSR are not the most efficient way to *transfer* data. To clarify, Figure 2.8 compares using a 2×3 systolic array for (i) multiplying a dense input matrix and a sparse weight matrix represented in CSR format (**A**); versus (ii) multiplying the same dense input matrix and the locally-dense weight matrix pruned by an *ideal* structured pruning algorithm suited for systolic architecture (**B**). Option A requires accessing extra metadata and reassembling a row before pushing it to systolic arrays, which has the following consequences: (i) In the worst case, the number of cycles is defined by the number of rows of the sparse matrix; (ii) At least one indexed read (i.e., the column) per non-zero value causes inefficient memory accesses. (iii) The compute units of the systolic array are poorly utilized. In contrast, in option B, the rows of the locally-dense weight matrix are streamed to the systolic array, without reading metadata. As a result, multiplications maximize bandwidth utilization with less hardware complexity.

The observation suggests that systolic arrays require optimization for *stream accesses to memory* not for memory footprint or the number of operations. As a result, even a pruning approach for *concurrent* hardware (e.g., SIMD) is not applicable for systolic arrays because

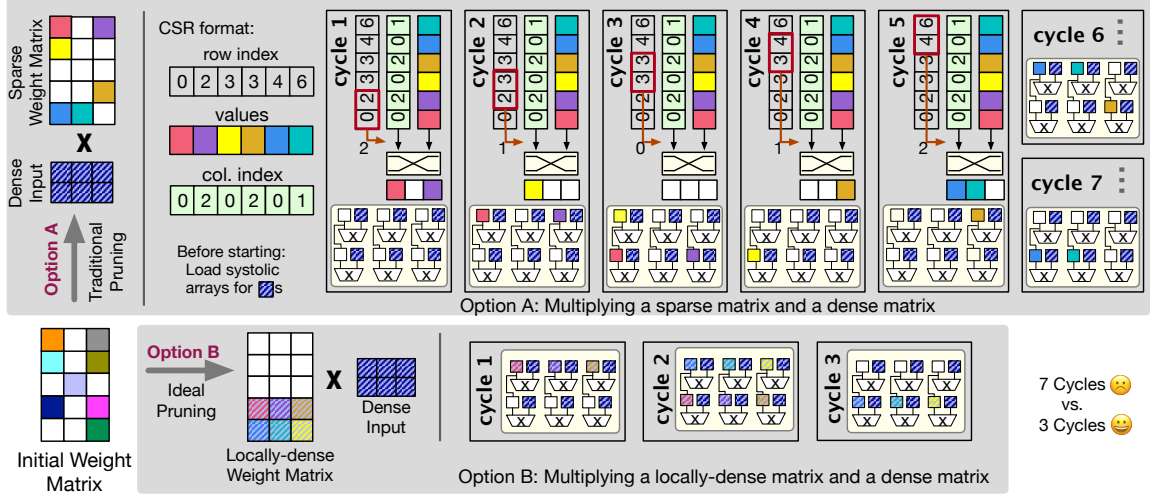


Figure 2.8: Clarifying the challenge of systolic-array underutilization: Comparing the execution of a matrix-matrix multiplication using a systolic array when one operand is a dense matrix and the other operand is: (a) a sparse matrix, and (b) a locally-dense sparse matrix.

they do not capture dependencies in data to satisfy the *data-reuse patterns*. Moreover, the computation results of the pruned model should be *compatible with the memory interface* (i.e., the stream interface) for eliminating extra buffering/caching. The storage adjacency of data resulting from algorithm-defined granularity data to the systolic array.

CHAPTER 3

PROCESSING SPARSE DATA WHILE GATHERING THEM

This chapter proposes a solution for the first category of sparse problems that are the problems in which data itself is dense but since at a moment only a small fraction of data is accessed randomly and irregularly, the pattern of accesses to memory is sparse. A main application of this category of sparse problems is recommendation systems that are the focus of this chapter.

3.1 Main Contributions of Fafnir

Our goal is to provide a solution to reduce data movement and provide memory and computation parallelism without relying on spatial locality. To achieve this goal, our key insight is to *process data while it is gathered* rather than processing data where it resides, mainly because in sparse gathering, data (i.e., different embedding vectors) do not reside in a single memory location; rather it is scattered. Based on this insight, we propose an efficient near-memory intelligent reduction (Fafnir*) [17], a data-centric solution for embedding lookup, which, unlike prior data-centric solutions, gradually applies the reduction on data while gathering them from random memory devices. The overhead for achieving the benefits explained in the following is $m - 1$ processing elements compared to prior work (that is, adding a 0.121 mm^2 at 7 nm chip).

Using an Overall Tree Structure: To apply reduction on embedding vectors from *any* memory devices without relying on spatial locality, we use an overall reduction tree, the leaves of which are connected to the ranks of a memory system and the nodes are reduction engines. In this way, we guarantee that all embedding vectors in a query are definitely reduced within the tree at NDP – it could occur in a leaf if the embedding vectors are from

*Fafnir is a star in the constellation of Draco.

neighboring memory devices or could occur at least at the root if the vectors are at the remotest locations. Therefore, while in all three schemes (i.e., TensorDIMM, RecNMP, and Fafnir) the mapping of vectors to DIMMs equally define the load of each NDP, only in Fafnir are the entire operations done at NDP, regardless of the mapping of vectors to DIMMs. Since Fafnir performs all the reduction operations at NDP, it guarantees to decrease data movement. In other words, similar to TensorDIMM, only the $n \times v$ elements corresponding to the outputs are transferred from the NDP to the cores. The tree structure of Fafnir also optimizes the number of connections. More specifically, instead of connecting NDP to the cores through the costly $c \times m$ all-to-all connections, Fafnir integrates computations within $2m - 2$ connections and then forwards them to the cores through c connections (i.e., total $(2m - 2) + c$). As a result of the fewer connections when adding more computation devices, Fafnir is also more scalable compared to prior work.

Parallelizing Memory Accesses & Computations: To fully utilize the tree and thus provide parallel computation while also reading data in parallel, Fafnir simultaneously activates distinct routes of the tree from arbitrary leaves to the root to process a batch of queries. Fafnir flows data corresponding to distinct queries through the tree in such a way that they do not conflict and hence their latency does not affect one another. As a result of this mechanism, Fafnir guarantees full utilization of the parallel computation at NDP (i.e., $n \times (q - 1) \times v$). Therefore, not only by *adding more ranks* to the system but also by *increasing the batch size* (i.e., processing more queries), we can better utilize the parallelism and achieve higher throughput.

No Caching Mechanisms: Fafnir uses a novel approach for processing a batch of queries with shared indices that do not require caching mechanisms for eliminating redundant accesses to memory. In other words, Fafnir reads only the unique indices from memory and then uses them as many times as required without storing data in a cache, hence preventing overhead such as searching, reading from, and writing to a cache. Fafnir rearranges a batch of queries and treats them as a set of unique indices. Therefore, Fafnir

accesses each unique index only once, and then, based on the query indices, it reduces the corresponding indices within the tree. Our observations, shown in Figure 3.1, illustrate the opportunity to effectively benefit from our novel batch processing mechanism (details in section 3.3). This mechanism of Fafnir also improves energy efficiency.

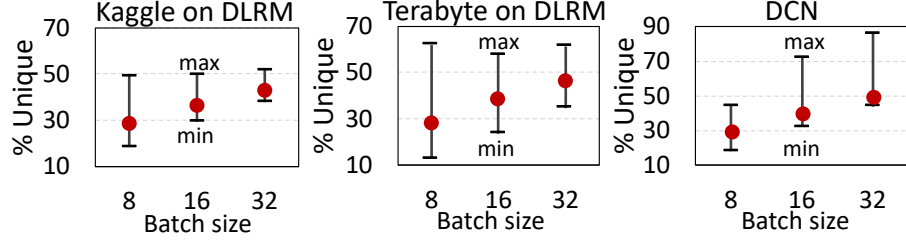


Figure 3.1: The percentage of unique indices in batches of queries.

Executing Various Sparse Problems: Customized hardware has not often been selected as a viable option. Instead, general-purpose hardware has usually been used for executing applications such as sparse problems, even though their performance is dramatically low. A reason for this is the economic aspect. Extensive customization has been expensive for narrow applications, even if such hardware offers significant performance benefits. To deal with the cost challenge, custom hardware solutions must be generic and applicable to a reasonable range of applications. To this end, we envision hardware for Fafnir that is generic enough to be used for executing other sparse applications, that include graph algorithms and scientific computations including matrix algebra, the main kernel of which is sparse matrix-vector multiplication (SpMV).

3.2 The Top-Down Overview of Fafnir

Software Support: Fafnir is a DDR-based NDP connected to a host for software support. The host is responsible for mapping data to the memory addresses, compiling the NDP kernels into a set of memory accesses, and calling Fafnir for executing NDP kernels by transmitting memory access requests to the root of the tree. The type of memory access differs based on the program. For instance, for embedding lookup, the host sends batches

of memory read addresses, whereas for SpMV (see section 3.4), it forwards stream accesses from all occupied ranks by specifying the initial memory address and the size of the stream. Besides, while for an embedding-lookup kernel the application software at the host arranges the queries, it performs vectorization for SpMV. The distribution of the memory accesses across the memory devices (ranks) is a result of the program behavior. The root receives the requests in the form of regular DDR4-compatible command/address (C/A) signals, decodes them, and forwards them to the corresponding (all if needed) DIMM/ranks across all parallel ranks. Ranks read data through DDR4-compatible data (DQ) signals and then all the special steps of Fafnir to gradually apply reduction operations from leaves to the root occur. Finally, the root sends the outcome back to the host.

Architecture: Figure 3.2a shows an overview of the Fafnir architecture, consisting of 32 ranks, and hence 31 processing elements (PEs), connected in a tree structure. In the current implementation of Fafnir, one leaf PE is connected to two ranks (i.e., 1PE:2R) and concurrently accesses them without creating conflicts by using the same techniques used in prior work [36, 35]. Similarly, depending on system requirements, other scales (e.g., 1PE:4R or 1PE:1R) are implementable. The PEs can be grouped as nodes in various ways. Each node would be a sub-tree of PEs, implemented in FPGA or ASIC. For instance, we can fabricate one PE chip of size $274\mu m \times 282\mu m$ at 7 nm (Figure 3.2a left layout) and embed it in a DIMM or put seven PEs together in a single $492\mu m \times 575\mu m$ chip to connect all the four DIMMs in a channel. In this paper, we implement two types of nodes: DIMM/rank and channel nodes. Accordingly, the Fafnir configuration consists of four DIMM/rank and one channel node. The nodes borrow their names from the source of their inputs. The input to each DIMM/rank node is from eight ranks (4 DIMMs, 2 ranks per each). A DIMM/rank node has seven PEs. the channel node has three PEs and its inputs come from four channels.

Figure 3.2b illustrates the mapping of embedding tables to 32 ranks of our target memory system – we map embedding vectors (e.g., each 512 bytes) to distinct ranks. Data flowing from leaves to the root of the tree includes a header and a value (the gathered

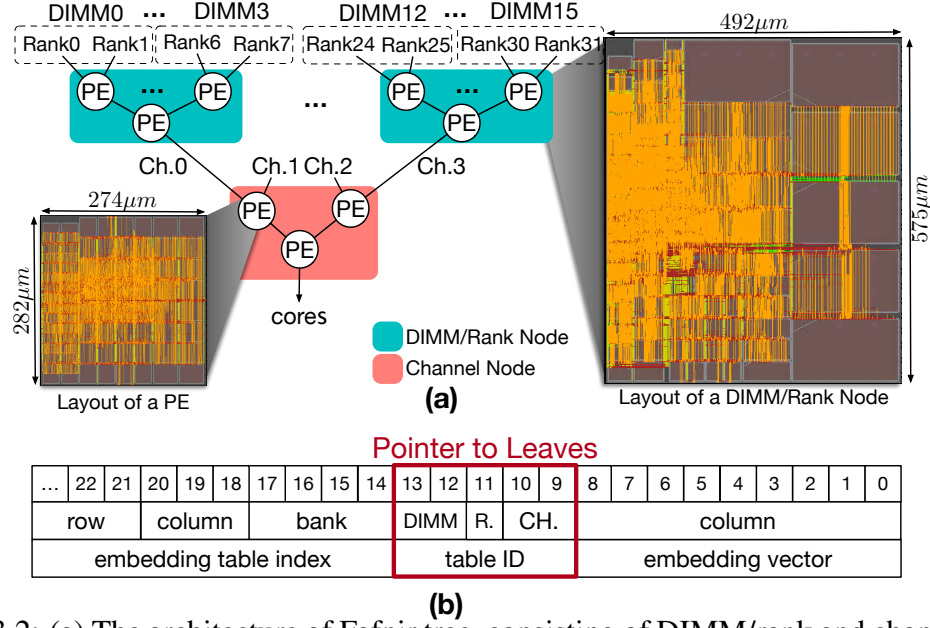


Figure 3.2: (a) The architecture of Fafnir tree, consisting of DIMM/rank and channel nodes and ASIC designs at 7 nm for a PE and a DIMM/rank node. (b) The mapping of embedding tables to memory addresses.

data). The header consists of two fields: *indices* and *queries*. The *indices* indicate the locations of memory from which data have been gathered (i.e, the bits [9-13] shown in Figure 3.2b). The *queries* indicate a list of indices for different queries that have not been visited, yet. For instance, assume that we have a query with indices **1, 2, 5, 6**, and data in a PE is the result of reducing data from indices **1, 2** and is yet to be reduced with data from indices **5, 6**. As a result, the output of that PE will have a header of **[indices:1,2|queries:5,6]**. By approaching the root and visiting more PEs of the tree, the indices from the *queries* field of the header are shifted to the *indices* field. Once data arrives at the root of the tree, the *queries* field will be empty, and the *indices* field will indicate a complete set of reduced indices for that query.

Microarchitecture of PEs: Figure 3.3 shows the microarchitecture of a PE, including two inputs (A and B) coming from a rank or the upstream PE in the tree architecture, and one output going to the downstream PE. A PE consists of two input FIFO buffers connected to compute units, the outputs of which are merged and directed to the output through a merge unit. The task of each PE is to process the headers and decide whether

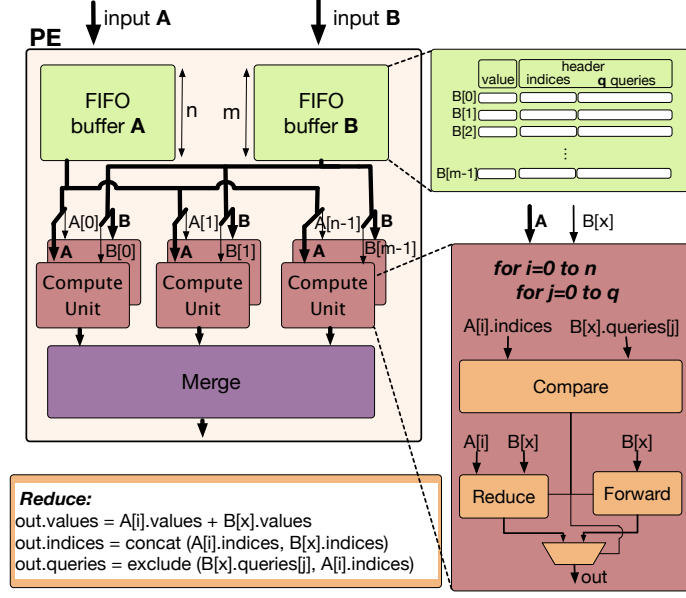


Figure 3.3: The microarchitecture of a PE including FIFO buffers, compute, and merge units, showing the data path from leaves to the root.

to reduce the inputs and assign a new header to it, or just forward them as they are. To enable processing a batch of inputs, we instantiate compute units, each of which iteratively compares one element of an input (e.g., $B[x]$) with all elements of the other input (e.g., A). More specifically, the entire *queries* field of $B[x]$ is compared with the *indices* field of $A[i]$ (i.e., $B[x].queries[j]$ and $A[i].indices$ are compared). If $B[x].queries[j]$ contains all elements of $A[i].indices$, the compute unit performs a reduction. If none of them match, it forwards $B[x]$. In each PE, we also compare the two inputs in the other way to make sure that the *queries* field of $A[i]$ is also matched with the *indices* field of $B[x]$. Since we process the inputs of a PE in parallel, the compute units may generate the exact same outputs concurrently, or multiple compute units may generate multiple outputs, the data of which are equal. In the first case, the redundant outputs must be removed, and in the second case, the outputs with the same data must be merged and the *queries* field in their headers must be merged (i.e., concatenated). Such post-processing is the task of the merge unit.

All PEs across the tree are identical. The size of the PE (i.e., the size of input buffers A and B and the number of computation units) could be tailored to better handle different

batch sizes. We define sizing based on the maximum size of inputs. Since we are processing batches of queries, each combination of the two inputs of a PE might be required by one of the queries in the batch. Therefore, in the worst case, a PE will need to generate all the possible combinations of its input to the output, which is a maximum of three combinations: Each of the inputs can individually be forwarded to output, or they can be reduced. Therefore, in theory, the number of outputs of a PE with two inputs of sizes n and m is $nm + n + m$.

The output size of a PE defines the input size of the downstream PE. As we move closer to the root, the size of the outputs and hence the size of consecutive inputs are supposed to be increasing, which demands larger buffers and more compute units. However, in fact, *the number of outputs of each PE is limited by the batch size*. This is simply because not all the combinations of the inputs are being used by a limited number of queries. While hardware is fixed for a batch size, larger batch sizes defined by software in various application domains are served as several small batches at hardware. Therefore, the maximum number of outputs for a PE is calculated as $\min(nm + n + m, B)$, in which B is the batch size. Table 3.1 lists the total size of buffers for PEs and nodes, which is the same for PEs at any level of the tree for three batch sizes. As Figure 3.3 shows, the buffers contain $n = m$ entries, each including a 512 B value and a 10 B header ($16 \times 5/8$) for $q = 16$ (i.e., each query includes maximum 16 indices) and 5-bit *indices/queries* fields for identifying embedding vectors from 32 embedding tables. In our configurations, $n = m = B$ also defines *the number of compute units* in a PE. When the size of inputs is smaller than the number of computation units, some compute units will simply have no value and remain idle.

Table 3.1: FIFO buffer sizes that are sum of all buffers in all PEs (B is batch size).

Node	PE buffer (KB)			Node buffer (KB)		
	$B = 8$	$B = 16$	$B = 32$	$B = 8$	$B = 16$	$B = 32$
DIMM/Rank Channel	4.6	9.3	18.5	32.4 13.9	64.8 27.8	129.5 55.5

3.3 Key Mechanisms

This section describes how Fafnir performs *eliminates redundant memory accesses* and *concurrent batch processing*. This mechanism, however, does not rely on batch processing. For instance, the same mechanism can also be used for interactive processing, in which all nodes would either forward or reduce without performing any comparisons. Fafnir first reads data corresponding to only the unique indices once, and then uses them within different queries as many times as required without using any caching techniques. The example in Figure 3.4 shows the steps of batch processing for a batch of four queries (i.e., a, b, c, and d) that access random embedding vectors from eight embedding tables and then processing them through a three-level tree, shown in Figure 3.4a. Only in this example, we assume that the indices to embedding vectors are created by concatenating the index within a table with a table number (e.g., **50** indicates index **5** from table **0**).

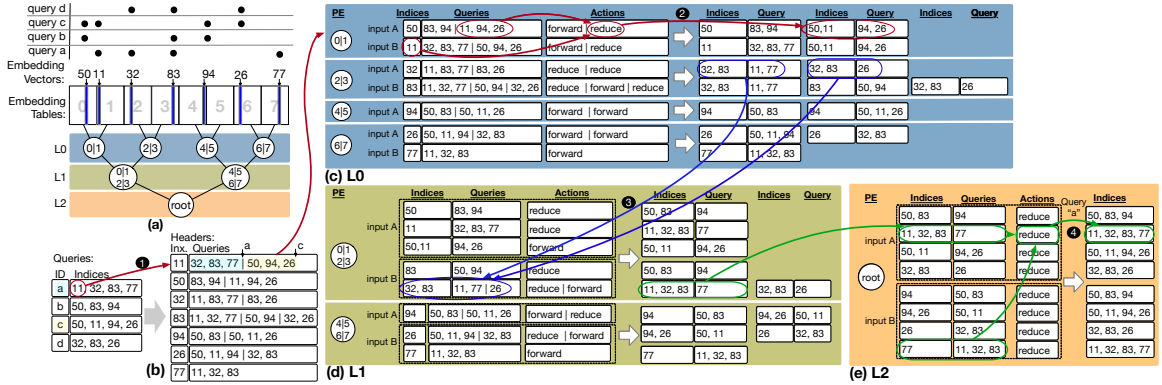


Figure 3.4: Concurrent batch processing and eliminating redundant memory accesses in Fafnir: (a) A batch of four queries that access random embedding vectors from eight embedding tables and a three-level Fafnir tree (b) Extracting the unique indices of four queries and creating the headers of requests to be forwarded to Fafnir. The steps of processing the four queries through the PEs at three levels of tree: (c) L0, (d) L1, and (e) L2.

To decrease the number of memory accesses, the host extracts the unique indices used in a batch of queries and creates the headers including *indices* (i.e., Inx) and *queries* fields (Figure 3.4b). To do so, the unique indices are added to the *indices* field. Then, all the indices of the queries, including that unique index but excluding the unique one, are added to the *queries* field. For instance, for the unique index **11** ❶, we add the following to the

queries field: **11, 32, 83, 77** from query a and **50, 11, 94, 26** from query c (**11** is excluded from both). In this way, instead of a total of 14 memory accesses, we access seven unique ones: **50, 11, 32, 83, 94, 26, 77**. The tables in Figure 3.4c, Figure 3.4d, and Figure 3.4e list the details of the processing steps at levels L0, L1, and L2 of the tree, respectively. These tables list (i) the headers of the input A and input B to each PE, (ii) the actions taken based on each comparison – each action corresponds to the result of comparisons of one item in the *queries* header, (iii) the header of raw outputs of each PE before merging, and (iv) the inputs to the next PEs, which are basically the merged outputs of the previous PEs.

PE **(0|1)** (similar to others) has two inputs, A and B. As the *queries* fields of A and B indicate, data from indices **50** and **11** will be used in two queries. In **(0|1)**, a compute unit compares item **[83, 94]** of A with the index of B (i.e., **11**) and since **11** is not included in **[83, 94]**, the compute unit forwards the value coming from input A, with its initial header of **[indices:50|queries:83, 94]**. Likewise, item **[11, 94, 26]** of A is compared with the index of B (i.e., **11**) and finds a match, thus reducing the values of A and B and creating the new header of **[indices:50, 11|queries:94, 26]**.

②. The *indices* field of the header is created by concatenating the indices of A and B and the *queries* field is created by excluding the indices of A and B from **[11, 94, 26]**. The compute units in PE **(0|1)** do the same for items **[32, 83, 77]** and **[50, 94, 26]** of input B, resulting in a forward and a reduce.

As Figure 3.4c shows, the initial outputs of PE **(0|1)** include the header **[indices:50, 11|queries:94, 26]** twice. In such a case, the merge unit is responsible for eliminating redundant outputs. The three unique outputs of PE **(0|1)** create the input A of PE **(0|1|2|3)**, the input B of which includes two items that have been created similarly in PE **(2|3)**. The number of initial outputs of PE **(2|3)**, however, is five. Besides the redundant outputs with headers **[indices:32, 83|queries:11, 77]** and **[indices:32, 83 |queries:26]**, PE **(2|3)** includes two groups of outputs with

the same indices **32, 83**, but different *queries* field. In such a case, the headers must be merged, because they are two headers for one unique value. The result of such merging is a value with the header of **[indices:32,83 | queries:11,77|26]** (shown in Figure 3.4d), which goes to input B of PE **(0|1|2|3) ③**. As the figure shows, because of merging, the size of input A and B never exceeds the batch size (i.e., four). The process of applying different actions on the inputs is similar in PE **(0|1|2|3)**, whereas here, each item of the *queries* field must be compared with the *indices* field of all items in the other input. Besides, as Figure 3.4c shows, in some cases, such as in PE **(4|5)**, only one of the inputs exists, which automatically leads to a forward action. By iteratively processing data and gradually reducing them (when required) through the tree, we reach the root PE, the outputs of which indicate the initial queries (Figure 3.4e). For instance, the green lines **(4)** show the final steps for creating query a.

3.4 Adapting Fafnir to SpMV

Sparse gathering is the *common* operation SpMV and embedding lookup, both of which can be implemented using a reduction tree. While this common feature allows adapting Fafnir to SpMV, maximizing the benefits for both requires addressing unique challenges that arise from their *differences*. The main difference between the *reduction* in embedding lookup and that in SpMV is that in embedding lookup, we reduce distinct vectors into one vector, whereas in SpMV, we need to reduce the elements of a vector into one element. Therefore, as Figure 3.5a illustrates, for an embedding lookup, each PE of Fafnir applies an element-wise reduction on two (or more) vectors and generates one output vector. For an SpMV, it is just the opposite: we need a reduction tree to sum the elements of a vector. As a result, the challenge is that if we simply use the reduction tree of Fafnir to execute SpMV, only one compute unit (reduce) of a PE will be utilized, as shown in Figure 3.5b. Our key insight to resolve this challenge is to use a *vectorization* technique along with an appropriate compression format. Figure 3.5c illustrates vectorization, in which each PE

processes a vector of independent elements of the sparse matrix and separately applies the reduction operation on them. Vector size could be the same as embedding-vector size.

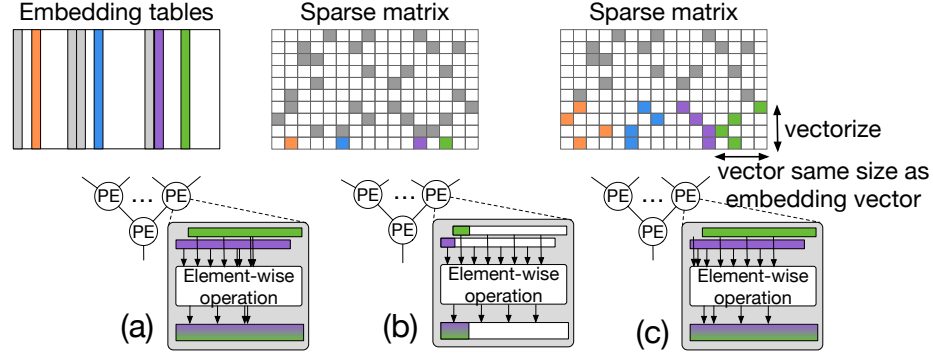


Figure 3.5: (a) Embedding lookup in Fafnir, (b) Using Fafnir for an SpMV with no mechanisms, and (c) Using vectorization to fully utilize Fafnir for SpMV.

Because of their differences, embedding lookup and SpMV use different mechanisms on the same hardware. However, if the primary application of Fafnir would be SpMV, the control logic shown in Figure 3.3 would be simpler because in SpMV, q is one and the iterations over q in compute units would not be necessary. Table 3.2 compares the mechanism of Fafnir for executing SpMV and embedding lookup. Unlike embedding lookup, for SpMV, the irregularity in memory accesses stems from *sparse data*. Because of such a difference, Fafnir handles memory accesses differently. First, as the second column of Table 3.2 lists, for SpMV, we do not know where the non-zero values of the sparse matrix are located. Therefore, when we read data from memory, the indices of the elements to be reduced are *unknown*. In fact, indices themselves are being read from memory. As a result, for SpMV, we stream both data and indices through the tree. Then, based on the indices, the tree reduces related values. In contrast, for embedding lookup, we know which indices we need to access. Therefore, we only stream data. The other difference between SpMV and embedding lookup is that the leaf PEs for SpMV first multiply data with the vector operands. The leaf PEs skip the multiplication for embedding lookup. Similar to embedding lookup, SpMV rather than caching mechanisms, uses a simple buffering, in which a vector operand is buffered in the multipliers until it is multiplied by matrix operand.

Table 3.2: SpMV vs. embedding lookup

	SpMV	Embedding lookup
Indices	Unknown	Known
Memory-access type	Stream data and indices	Stream data only
Leaf PE	Multiplication with vector	Skip multiplication

To facilitate streaming sparse matrices, we suggest using the list-of-list (LIL) compression format, which has become popular in recent sparse studies [68, 111, 112] (sometimes called other names such as linked list). Further, LIL is supported by the SciPy library, which makes its application more straightforward. LIL compresses the non-zero values of the original sparse matrix in one dimension and saves the indices corresponding to the other dimension of the matrix. As LIL compresses matrices only in one dimension, it facilitates splitting large matrices into chunks through their non-compressed dimension, hence facilitating parallel streaming. The ease of splitting and parallel streaming is important in large sparse matrices (e.g., graph problems or HPC). To apply SpMV on large matrices that do not fit into Fafnir, we split them through their non-compressed dimension. Similar splitting is also used in the state-of-the-art NDP approach for SpMV [76].

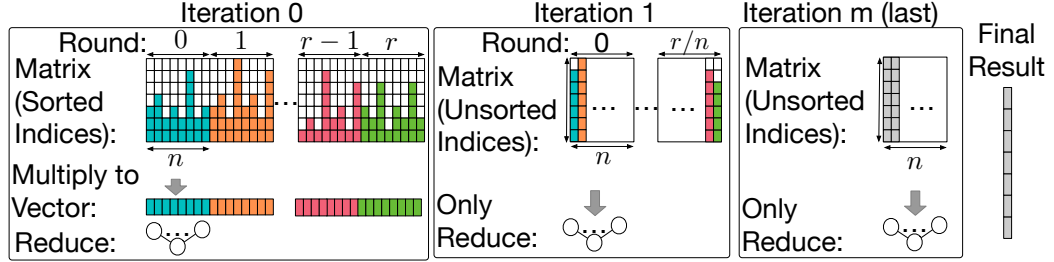


Figure 3.6: The iterations and rounds for SpMV on large sparse matrices using Fafnir when only n columns of the matrix fits to Fafnir at a time.

As Figure 3.6 shows, we perform an SpMV in iterations, each consisting of several rounds. In the first iteration (iteration 0, which is functionally equivalent to the first step of Two-Step algorithm [76]), the matrix is multiplied by the vector operand, whereas all other iterations only merge the results of the previous iteration. We use the same hardware (DIM-

M/rank and channel nodes of Fafnir) for both types of iterations. During merge iterations (i.e., iterations > 0), leaf PEs skip the multiplications as they do in embedding lookup. In addition, during the merge iterations, the row indices are no longer sorted, but this does not impact the functionality of Fafnir. Figure 3.7 illustrates the number of required iterations and rounds per iterations for two vector sizes (i.e., 1024 and 2048) when the number of columns (and rows) increases up to 20 million. As the figure suggests, even for matrices with more than 5 million columns, no more than two merge stages are required.

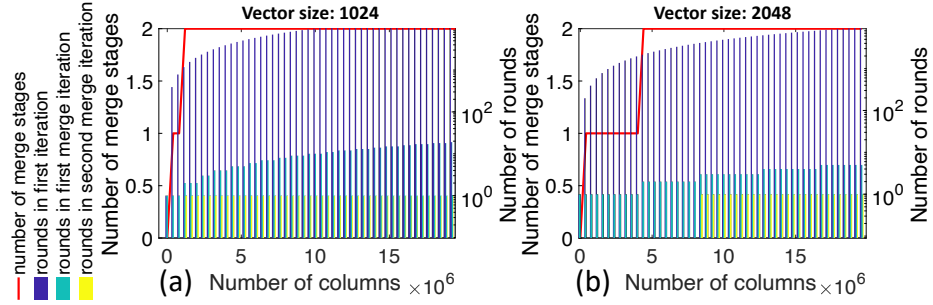


Figure 3.7: The number of iterations, rounds per iteration, and required merges for matrices with up to 20 million columns, for vector sizes (a) 1024 and (b) 2048. In our configuration for SpMV, vector size (i.e., the number of columns that fit in Fafnir tree) is 2048.

3.5 Evaluation

3.5.1 Experimental Setup

Figure 3.8 shows an overview of our design, implementation, and evaluation flow. We implement the microarchitecture of Fafnir (and the baselines) in C++. We use our hardware description in C++ for (i) RTL generation and subsequently FPGA and ASIC implementation, and (ii) performance evaluation. To generate RTL (in Verilog), we use related `#pragmas` as hints to describe the microarchitectures. We use Vivado HLS to generate RTL and Vivado to synthesize and implement our design on an XCVU9P FPGA, targeting a VCU1525 acceleration development kit, which includes four 16 GB DDR4 DIMMs (64 GB total per DIMM/rank node). Besides reporting resource utilization and power consumption for FPGA, we implement the ASIC design of Fafnir using the toolchain of Synopsys design

compiler (DC), Cadence Innovus, and Cadence Tempus. As an input to our ASIC design, we use our same Verilog code generated by HLS and just substitute the BRAM blocks with memory cells. Our ASIC design is based on an Arizona State Predictive PDK (ASAP) 7nm technology node [113], a free PDK for non-commercial academic use. All performance numbers reported in this paper are based on FPGA-based C/RTL co-simulation results (as shown in Figure 3.8). For verifying functionality at scale we perform regression testing using large synthetic data through a C++ testbench for C/RTL co-simulation. To facilitate performance evaluation for large real-world data, we inject the FPGA post-implementation timing analysis @200MHz into our C++ emulator, the core description of which is initially used for RTL generation.

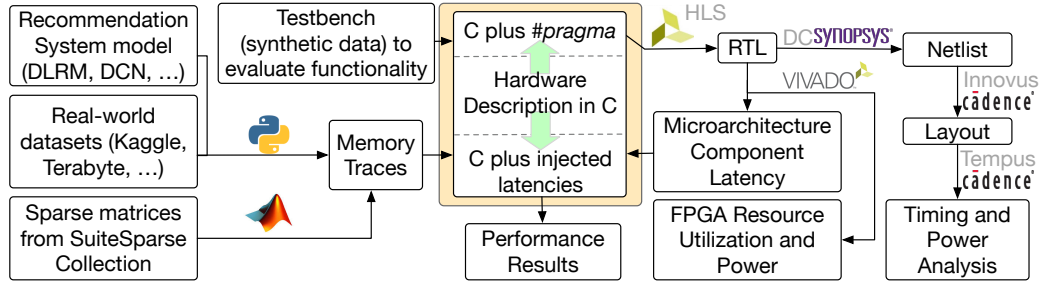


Figure 3.8: Experimental setup.

We evaluate two applications: (i) recommendation systems including embedding lookup and (ii) graph analytics and scientific applications, both including SpMV. For scientific applications, we execute a matrix inversion algorithm (the most bottleneck-prone algorithm) using the lower-upper technique, which also iteratively calls SpMV. The inputs to our C++-based emulator are memory traces based on accesses to embedding tables of recommendation systems and the sparse matrices for SpMV-based applications. For the recommendation systems, we run Deep & Cross Network (DCN) [114] as well as Deep Learning Recommendation Models (DLRM) [13] based on two real-world open-source data sets: (i) the Criteo Ad Kaggle data set [115] containing approximately 45 million samples over seven days and (ii) the Criteo Ad Terabyte data set [116] sampled over 24 days. We logged the indices of embedding-table accesses and preprocess them using Python scripts to gen-

erate memory-access traces. To prepare the inputs for SpMV-based applications, we use Matlab to preprocess our sparse matrices, listed in Table 3.3, obtained from the SuiteSparse collection [117], six from the scientific-computing domain and six graphs.

Table 3.3: Sparse matrices from SuiteSparse [117].

ID	Name	Dim.(M) ¹	Density (%)	Application
RE	N_reactome	0.016	0.025	Biochemical
RI	rail582	0.056	1.2	Linear Prog.
HC	hcircuit	0.1	0.004	Circuit Sim.
2C	2cubes_sphere	0.101	0.016	Electromagnetic
TH	thermomech_dK	0.2	0.006	Thermal
FR	Freescall2	2.9	0.0001	Circuit Sim.
AM	amazon0601	0.4	0.002	Dir. Graph
WG	web-Google	0.91	0.0006	Dir. Graph
RO	roadNet-TX	1.3	0.0001	Unidir. Graph
KR	kron_g500-logn21	2	0.004	Unidir. Multigraph
WI	wikipedia-20070206	3.5	0.0003	Dir. Graph
LJ	soc-LiveJournal1	4.8	0.0002	Dir. Graph

¹ Dim.: dimension or the number of columns/rows of a square matrix.

Our baseline NDP designs for embedding lookup are TensorDIMM [35] and Rec-NMP [36], and for SpMV-based applications is the Two-Step algorithm [76]. The Two-Step [76] algorithm is the state-of-the-art NDP accelerator for SpMV, which converts random memory accesses to regular accesses and ensures full memory streaming. The Two-Step algorithm mostly focuses on optimizing the implementation of the merge step (i.e., iterations > 0 in Figure 3.6) by using a binary tree-based multi-way merge core. The main contribution of the Two-Step algorithm is parallelizing the multi-way merge operation to handle *large* and *highly sparse* graphs. To reproduce the performance numbers of preceding NDP accelerators, we implement them on our FPGA platform based on the information/configurations provided in their published papers. We validate the reproduced numbers against their reported numbers.

3.5.2 Latency

Table 3.4 lists the latency of the compute-unit components that define the latency of pipeline stages and the critical path for our FPGA implementation @200MHz. The critical-path latency is defined by the latency of the compare and reduce units (since reduce and forward are parallel and reduce is slower). Before investigating the key metrics that are end-to-end speedup, scalability, and energy, we quantitatively compare the single-query latency of Fafnir with baselines. To do so, we measure the latency of a query, including random accesses to 16 512B vectors distributed over 32 ranks ($4 \times$ channels, $4 \times$ DIMMs, $2 \times$ ranks).

Table 3.4: Latency (cycles @200MHz) of the components in compute units of Fafnir for FPGA implementation.

	Compare	Parallel paths (reduce or forward)			
		Reduce (value)	Reduce (header)		Forward
			indices	queries	
per item (iteration)	12	3	4	3	16
batch size = 8/16/32	N/A		32/64/128	29/53/101	N/A

Figure 3.9 shows the contribution of memory access and computation (reduction operation) in total latency. Several parameters such as vector size and number in a query, row buffer size, distribution of vector, the number of pipeline stages, and DRAM timing define the effectiveness of (i) benefiting from row-buffer locality and (ii) not relying on spatial locality, on computation and memory latency. For instance, as Figure 3.9 illustrates, the computation latency of TensorDIMM, which *pipelines* the processing of 16 embedding vectors in a query, is $2.5 \times$ slower than Fafnir, which processes all 16 vectors in *parallel*. Although the parallelism level of RecNMP is also similar to that of Fafnir, its computation latency is not as low as in Fafnir because RecNMP forwards a few (here $\sim 25\%$) computations to the CPUs as a result of lack of spatial locality. In terms of memory latency, however, Fafnir and RecNMP are identical since they similarly utilize rank-level parallelism and row-buffer hit. In this example, the memory latency of TensorDIMM is $4.45 \times$ slower than RecNMP and Fafnir, which could have been up to $16 \times$ slower in the case of no row buffer hit.

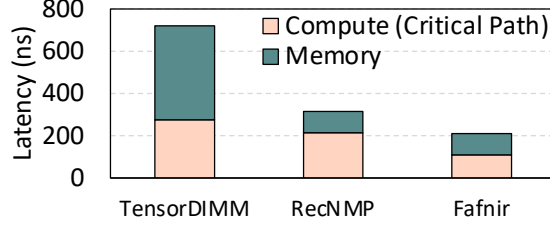


Figure 3.9: Single-query latency breakdown.

3.5.3 Speedup and Scalability

To evaluate the impact of accelerating the embedding lookup on the overall inference latency, Figure 3.10 shows the end-to-end speedup of RecNMP and Fafnir over the baseline (1-rank) when increasing ranks from two to 32. The figure shows the breakdown of total inference latency into three components: (i) embedding lookup; (ii) fully-connected (FC) layers executed at CPU, the performance of which is assumed to be fixed in various ranks. In Figure 3.10, FC layers take 0.5 *ms*, however, their latency varies significantly based on the host system (CPU vs. GPU) and batch size [12] – optimizing the performance of FC layers is not the focus of this paper; and (iii) other operations. While both RecNMP and Fafnir work close to the ideal linear speedup (red line) for fewer ranks, Fafnir keeps following the red line more closely as the number of ranks increases to 32. This stems from the key difference between RecNMP and Fafnir: the DIMM-level parallelism by putting a small chip (channel node) between memory and the core to perform *all* reductions at NDP rather than in the cores, the impact of which is more pronounced in larger memory systems with more ranks.

We also evaluate the speedup of Fafnir over the state of the art for two SpMV-based applications. While Fafnir performs the first step (iteration 0) of SpMV more quickly, the Two-Step algorithm more quickly merges the result (iterations >0). This is because, unlike the Two-Step algorithm, Fafnir does not rely on decompression mechanisms and is able to apply SpMV on data as it is streamed from memory. Further, instead of a chain of adders connected to multipliers, Fafnir uses the tree for the reduction. Conversely, since the Two-Step algorithm particularly optimizes the merge operation, it performs the merge steps more

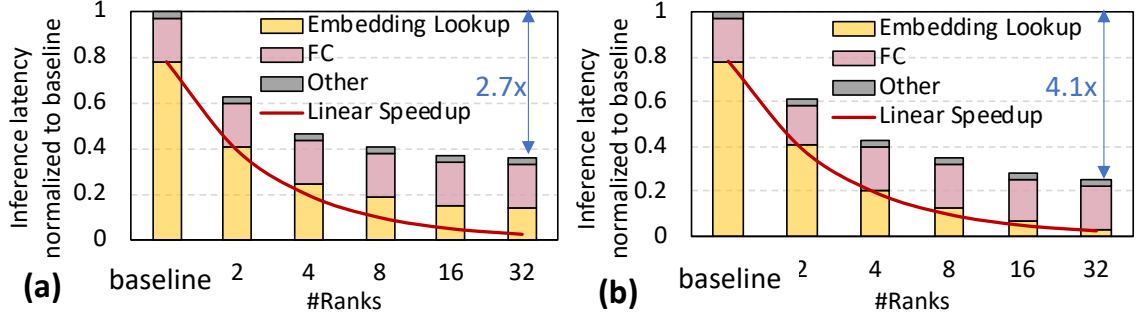


Figure 3.10: End-to-end inference speedup for DLRM on Kaggle (batch size = 8): (a) RecNMP, (b) Fafnir.

quickly. Because of the mentioned reasons, as Figure 3.11 illustrates, with no modifications in hardware, Fafnir can process SpMV-based sparse problems more quickly (e.g., up to $4.6\times$) or in the worst case as quickly as (e.g. $1.1\times$) the Two-Step. For smaller matrices, as fewer merge iterations are required, Fafnir performs more quickly than larger ones. In some workloads among the larger matrices (e.g., RO) sparseness is a reason that makes them more suitable for Fafnir. Based on our observation, a promising future direction is the combination of both Fafnir (for the first step) and Two-Step (for merging).

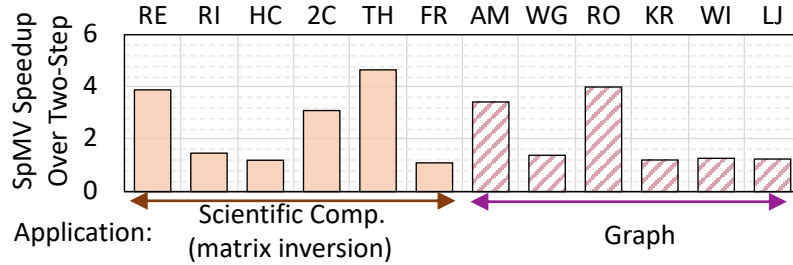


Figure 3.11: Speedup of Fafnir over Two-Step algorithm [76] for two SpMV-based applications: scientific computations (matrix inversion algorithm) and graph.

In a scalable design, increasing the batch size must help increase throughput. To evaluate the impact of concurrent batch processing on scalability, Figure 3.12 illustrates the speedup over RecNMP when batch size varies. Although all three designs utilize batch processing, their difference is in the hardware mechanism to most effectively take advantage of a batch to improve throughput. As Figure 3.12 illustrates, RecNMP looks up embedding approximately $15\times$ faster than TensorDIMM. This speedup stems from the approach of RecNMP to utilize rank-level parallelism. As Figure 3.12 shows, the speedup of Fafnir

over RecNMP, however, more significantly grows with the batch size (i.e., $3.1\times$, $6.7\times$, and $12.3\times$, for batch size 8, 16, and 32, respectively) when neither Fafnir nor RecNMP eliminates redundant memory accesses. The reason is that Fafnir better utilizes memory bandwidth, therefore, filling the gap under the roofline model of RecNMP by performing full-reduction near memory. The tiny (i.e., 0.121 mm^2) channel-node chip between the memory channels and core is the key to achieve this. In addition, as the striped part of Figure 3.12 shows, Fafnir achieves up to an extra $3.4\times$ speedup by more effectively eliminating redundant accesses to memory without using caches. For RecNMP, we assume 128KB rank caches that offer the optimal hit rate of 50%.

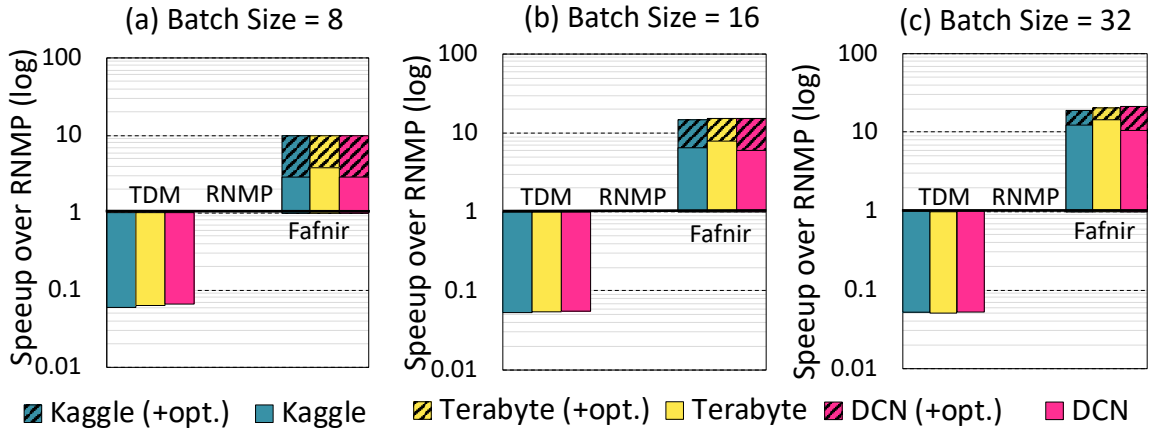


Figure 3.12: Speedup of Fafnir and TensorDIMM [35] (TDM) over RecNMP [36] (RNMP) for batch sizes (a) 8, (b) 16, and (c) 32. Opt. stands for the optimization of elimination of the redundant memory accesses.

3.5.4 Power Consumption and Area Overhead

This section evaluates the hardware of Fafnir (assuming $n = m = 32$ in Figure 3.3 and 32 compute units at PEs). Table 3.5 lists the resource utilization of Fafnir implementation on FPGA. To embed Fafnir in a standard DIMM-based memory system including four channels, each with four DIMMs, including two ranks, we need four DIMM/rank nodes and one channel node. The implementation of such a system utilizes up to 5%, 0.15%, 1%, and 13% of LUTs, LUTRAMs, FFs, and BRAM blocks of the target FPGA. Figure 3.13a shows the breakdown of dynamic power consumption of FPGA @200MHz, in total 0.23 W

and 0.18 W for DIMM/rank and channel nodes. For our ASIC design, Figure 3.13b shows the power distribution of a PE. As shown, power consumption has a uniform distribution, which prevents the creation of a hot spot. As well, the breakdown of the power consumption of our ASIC design is listed in Table 3.6. Our proposed chips add only 23.82 mW per four DIMMs (i.e., 5.9 mW per DIMM) and in total, 111.64 mW to a four-channel memory system, which is negligible compared to the 13 W power consumed by each DDR4 DIMMs, calculated based on a Micron power calculator [35, 118]. As another comparison point, a processing unit RecNMP [36] adds 184.2 mW to one DIMM (at 40 nm @ 250 MHz).

Table 3.5: FPGA resource utilization for Fafnir.

Resources	DIMM/Rank Node		Channel Node	
	units	utilization(%)	units	utilization.(%)
LUT	11800	1.00	7214	0.61
LUTRAM	192	0.03	96	0.02
FF	4646	0.2	3295	0.14
BRAM	68	3.15	26	1.2

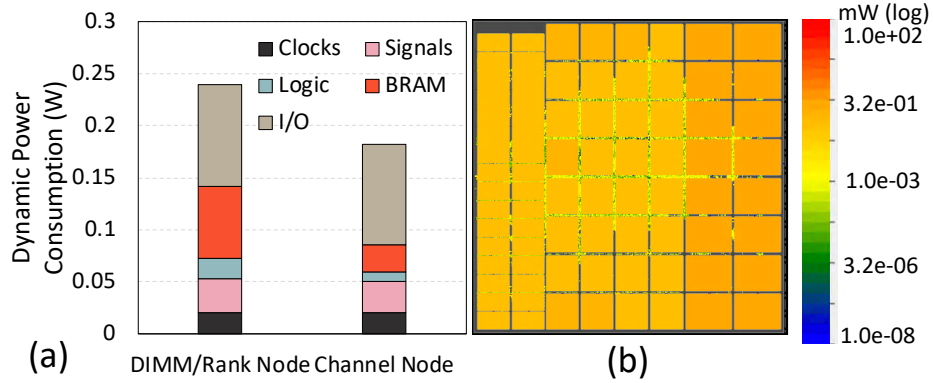


Figure 3.13: (a) Dynamic power consumption breakdown of Fafnir on FPGA. (b) Power distribution of a PE in our ASIC design at 7 nm.

Table 3.6 lists the area of PE and two types of nodes in Fafnir. A PE is 0.077 mm^2 (including the multiplication units for leaf PE to support SpMV) and the area of DIMM/rank and channel nodes is 0.282 mm^2 (which is smaller than the 0.077×7), and 0.121 mm^2 , respectively. Therefore, a benefit of embedding PEs into one chip (as we do) rather than distributing them across DIMMs is a more efficient area. Based on these numbers, we add

a total area overhead of 1.2 mm^2 to a memory system of 32 ranks. As a comparison point, the area of prior work, a RecNMP [36] processing unit, is estimated as 0.54 mm^2 at 40 nm per one DIMM (8.64 mm^2 to entire 16 DIMMs).

Table 3.6: Area and power consumption breakdown @500MHz to switching (Sw.), inter-connections (Int.), and leakage (Lkg.) for ASIC design of Fafnir @7 nm.

		PE	DIMM/Rank	Channel
Power(mW)	Sw.	2.1	3.6	2.7
	Int.	8.7	20.1	13.61
	Lkg.	0.02	0.12	0.06
Area(mm^2)		0.077	0.282	0.121

Given that the energy consumption of DRAM dominates that of computation, the energy-savings of memory is essential. Fafnir promises memory energy savings by eliminating extra memory accesses *without using any caching mechanism*. More specifically, Fafnir saves 34%, 43%, and 58% memory accesses for batch sizes 8, 16, and 32, respectively. Figure 3.14 illustrates the number of memory accesses after eliminating redundant accesses and shows that the number of memory accesses per each input to the leaf PEs is always lower than the batch size (8, 16, and 32 in Figure 3.14a, Figure 3.14b, and Figure 3.14c).

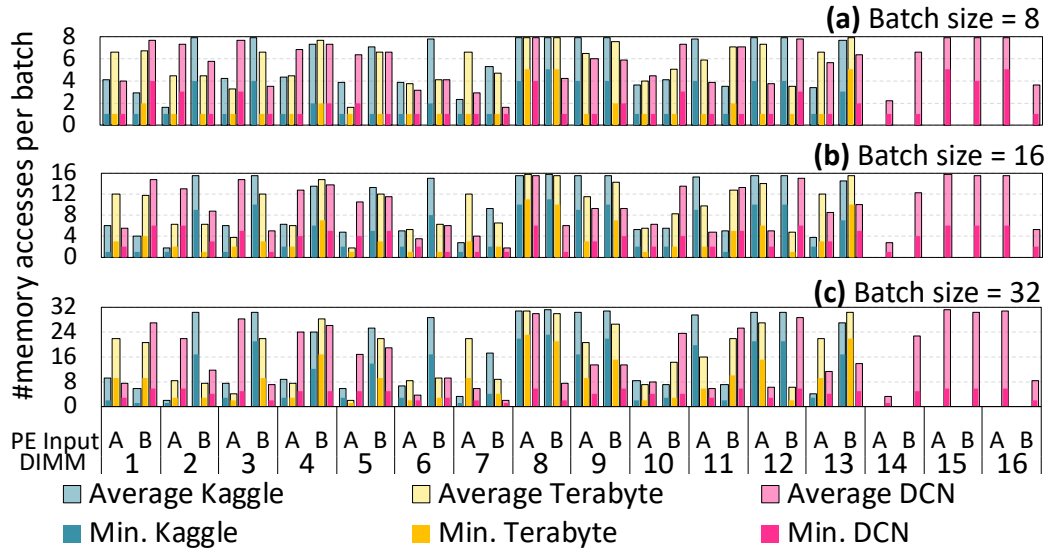


Figure 3.14: Number of memory accesses at different DIMMs for three batch sizes:(a) 8, (b) 16, and (c) 32.

3.6 Summary

In this chapter, we saw that memory-bound sparse gathering, caused by irregular random memory accesses, has become a challenge in several on-demand applications such as embedding lookup in recommendation systems. To address the challenge of data movement, prior work has proposed NDP solutions. However, we saw that prior work, either minimize data movement effectively at the cost of limited memory parallelism or improve memory parallelism but cannot successfully reduce data movement. A reason for that is relying on spatial locality, which is an unrealistic assumption to utilize NDP. More importantly, neither approach proposes a solution for sparse *gathering*; rather they just offload operations to NDP. This chapter introduced Fafnir, an effective solution for *sparse gathering*, an efficient near-memory intelligent reduction tree, the leaves of which are all the ranks in a memory system, and the nodes gradually apply reduction operations while data is gathered from *any* rank. Since Fafnir uses an overall tree, it performs the *entire* operations at NDP and fully benefits from parallel memory accesses in parallel processing at NDP. Further, this chapter showed that Fafnir offers other advantages such as using fewer connections because of the tree topology, eliminating redundant memory accesses without using any caching mechanisms, and being applicable to other domains of sparse problems such as scientific computing and graph analytics. Our evaluation results based on an XCVU9P Xilinx FPGA and in 7 nm ASAP ASIC showed that Fafnir looks up the embedding tables up to $21.3\times$ more quickly than the state-of-the-art NDP proposal. Furthermore, the generic architecture of Fafnir allows running classic sparse problems using the same 1.2 mm^2 hardware up to $4.6\times$ more quickly than the state of the art.

CHAPTER 4

MATHEMATICAL TRANSFORMATION TO REDUCE DEPENDENCIES

Even if by using Fafnir we are able to transfer sparse data quickly and efficiently, we still need to answer another question: what if computation creates a bottleneck? This chapter answers this question by focusing on scientific computing and solving PDEs, another category of sparse problems, in which computations creates a bottleneck. Our key insight to efficiently accelerate the iterative solvers of PDEs is to reduce the negative impact of data dependencies on performance by hardware-software co-design, even though we cannot remove the patterns of data dependencies that naturally exist in a program. In our example case, for instance, to allow the unrolling and blocking to be effective, steps 2, 3, and 4 must be executed quickly, preferably in one step as shown in Figure 4.1. We propose Alrescha* [18], a hardware-software co-design that divides a SymGS into a large portion of general matrix-vector multiplications (GEMVs) that can be executed in parallel or concurrently and a small data-dependent SymGS. Since the SymGS part is now small, Alrescha [18] can execute it quickly in hardware. To be effective in fast execution of SymGS, Alrescha accelerates (i) the mechanism of immediately using the outcome of operation by the next operations in the SymGS (\bar{x}_i i, ❶); and (ii) the mechanism of using the outcomes of the GEMV in different operations of SymGS (x'_i , ❷).

4.1 Key Mechanisms of Alrescha

After dividing a large SymGS into GEMVs and a small SymGS, the proposed hardware mechanisms of Alrescha help to execute them quickly. To explain the mechanisms, we use a simple example of a SymGS with matrix A operands shown in Figure 4.2a. We focus on processing three rows of A ($i = 4, 5, 6$) to calculate corresponding final elements of x

*A binary star, the two stars of which orbit one another.

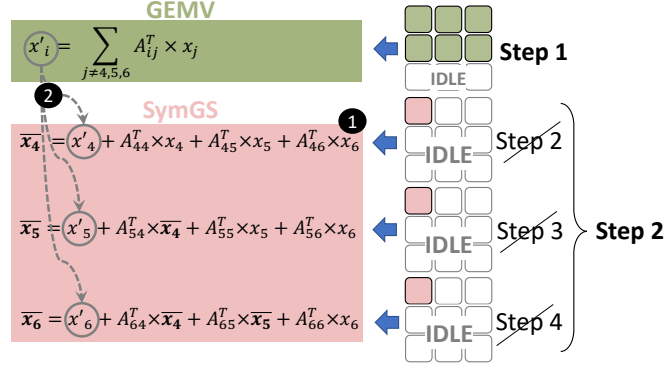


Figure 4.1: Key insight of Alrescha: We divide a large SymGS into a majority of parallelizable GEMV operations (green) and a minority of *small* data-dependent SymGS (pink). We first run the GEMV and then switch to SymGS. Dependencies still exist in SymGS part but as long as it is small, we can run them in one step in hardware rather than three steps (❶). Forwarding the outcomes of GEMV to the SymGS must be fast as well (❷).

(i.e., \bar{x}_4 , \bar{x}_5 , and \bar{x}_6). The green parts of matrix A are the operands to GEMVs, while the pink part is the operand of the small SymGS.

Parallel & concurrent GEMVs: First, Alrescha executes all GEMV operations that result in the partial outputs (i.e., x'_4 , x'_5 and x'_6). Figure 4.2b, Figure 4.2c, and Figure 4.2d show last three steps of the GEMVs that contribute in creating x'_6 , x'_5 and x'_4 , respectively. Such an order of operations that first performs all the GEMVs corresponding to rows 4 to 6 before performing the SymGS corresponding to the same rows, does not impact the functionality and the correctness of the operations as long as the partial results are correctly aggregated with corresponding values in the next steps, as explained in the following.

Fast switch from GEMVs to SymGS: Alrescha facilitates the mechanism of aggregating the partial results generated by GEMVs with the partial results of SymGS by using a last-in-first-out (LIFO) – alternatively, a first-in-first-out (FIFO) buffer can also be used if compatible orders also reflected in reading the rows of the matrix. As Figure 4.2b, Figure 4.2c, and Figure 4.2d show, during GEMV phase, we push the partial results into the LIFO, and POP them out during SymGS, as illustrated in Figure 4.2e, Figure 4.2f, and Figure 4.2g. This mechanism, which prevents extra accesses to an on-chip cache (with sophisticated addressing requirements) or the main memory, provides a smooth switching

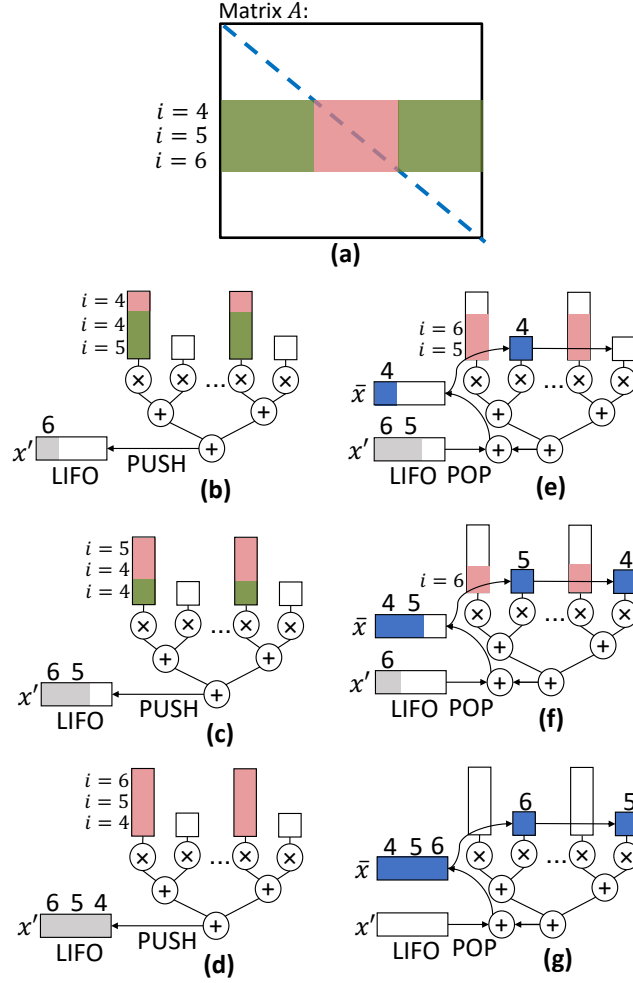


Figure 4.2: Key mechanisms of Alrescha: (a) Matrix A the operand of the original large SymGS, (b, c, d) Executing GEMV and the mechanism for quickly forwarding the outcome of GEMV to SymGS using a LIFO or FIFO; and (e, f, g) Executing the small SymGS and implementing the scheme of data dependencies through the interconnections between inputs of the tree and the LIFO to quickly execute the small SymGS.

between GEMVs and SymGS operations.

Fast execution of data-dependent SymGS: Once all the GEMVs corresponding to rows 4 to 6 are done, Alrescha switches to SymGS (the new step 2 in Figure 4.1). The nature of SymGS in step 2 is the same as the GEMVs in step 1, whereas the individual inputs are not available altogether. In fact, step 2 is generating its own inputs. Because of the similarity between the GEMV and SymGS, SymGS can use the same core mechanism of multiplication followed by the summation-based reduction tree as shown in Figure 4.2e,

Figure 4.2f, and Figure 4.2g. Besides the core mechanism, Alrescha implemented the dependencies using some interconnections between the inputs of the tree and its output. Such interconnection immediately forwards \bar{x}_i to the inputs and shifts the old inputs to the right. this mechanism simply accelerates the three dependent operations in old steps 2, 3, and 4. Note that the smallness of SymGS block is important here since otherwise, the depth of the tree prevents the fast execution.

4.2 Compression Format for Sparse PDEs

As explained earlier, the matrix A in a linear system is often *sparse*. Therefore, a compression format must be used to efficiently save the matrix A . On the other hand, we saw that the hardware mechanisms demand a unique order of data in matrix A . This section discusses the compression formats suitable for the target applications and explains how we slightly modify an appropriate compression format to sustain the desired order of data, dictated by our proposed mechanism. According to the distribution of non-zeros in a sparse matrix, various compression formats may suit them. For instance, the compressed sparse row (CSR), which stores a vector of column indices and a vector of row offsets, locates all the non-zeros independently is the right choice when the non-zeros do not exhibit any spatial localities. On the other hand, when all the non-zeros are located in diagonals, the diagonal format (DIA) [110], which stores the non-zeros in the diagonals sequentially, could be the best option. An extension to the DIA format, Ellpack-Itpack (ELL) [107] is more flexible when the matrix has a combination of non-diagonal and diagonal elements. For instance, ELL is used for implementing SymGS in GPUs. However, such a format does not provide flexibility for parallelizing rows as it does not sustain locality across rows.

Since the choice of compression format should be compatible with the range of sparse applications, blocked CSR (BCSR) [87], an extension of CSR, which assigns the column indices and row offsets to blocks of non-zero values, has been proposed as a more generic format. Although BCSR is an appropriate format for scientific applications and graph

analytics in terms of storage overhead, the strategy of BCSR for assigning indices and pointers, and the order of values, is not the most appropriate match for smoothly streaming data in Alrescha. In other words, the main requirement for fast computation is the order of operations, which in turn, dictates the data structures to be streamed in the same order. Thus, we adapt BCSR and propose a new compression format with the same meta-data overhead but compatible with Alrescha.

Figure 4.3 illustrates our proposed compression format for mapping an example sparse matrix to the physical memory addresses of the accelerator. In this compression mechanism, all the non-diagonal non-zero blocks in a row of blocks are stored sub sequentially, followed by a diagonal block. The non-zero values belonging to the upper triangle of the non-diagonal blocks are stored in the opposite order of their original locations in the matrix (see the order of A, B, and C in Figure 4.3). Accordingly, the difference between the column indices of BCSR and input indices (i.e., Inx_{in}) of our proposed format is shown in Figure 4.3. For SymGS, the diagonal of A is excluded and stored separately in a local cache. Therefore, we consider non-square blocks on the diagonal (e.g., 3×4 instead of 3×3) so that the mapping of the non-diagonal element of that block to the physical memory is adjusted. The indices of the input and output (i.e., Inx_{in} and Inx_{out}) are not streamed from memory during run time. Instead, they are stored in a configuration table during a one-time programming phase and are used for reconfiguration purposes. As a result, during the iterative execution of the algorithms, the whole available memory bandwidth is utilized only for streaming payload.

4.3 Broad Applications

To deal with the high design and fabrication costs cost of customized hardware, we argue that custom hardware solutions must be generic and applicable to a reasonable range of applications. This section elaborates on the applicability of Alrescha for SpMV and graph analytics. In graph analytics, a common approach to represent graphs is to use an adjacency

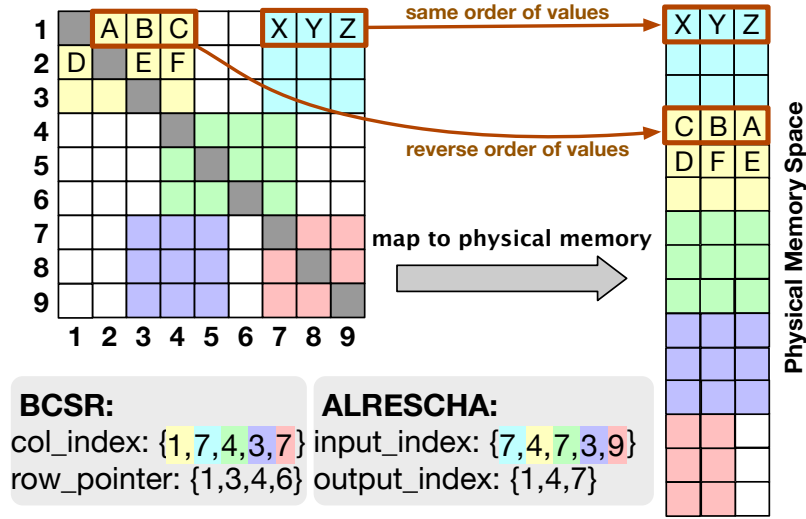


Figure 4.3: Compression format of Alrescha: the col_index of BCSR and input_index (i.e., Inx_{in}) of Alrescha are color-coded to show their corresponding blocks in the matrix. Alrescha uses the index of the last column for the input index of diagonal blocks.

matrix, each element of which represents an edge in the graph. Graph algorithms traverse vertices and edges to compute a set of properties based on the connectivity relationships. Traversing is implemented as a form of a dense-vector sparse-matrix operation. Such implementations are suited to the vertex-centric programming model [119], which is preferred to the edge-centric model. The vertex-centric model divides a graph algorithm into three phases. In the first phase, all the edges *from* a vertex (i.e., a row of the adjacency matrix) are processed. This process is a *vector-vector operation* between the row of the matrix and a property vector, varied based on the algorithm. In the second phase, the output vector from the first phase is *reduced* by a reduction operation (e.g., sum). In the final phase, the result is assigned to its destination. Since in many applications not all the nodes in a graph are connected, the equivalent adjacency matrix is sparse, too.

The widely used graph algorithms are SpMV, BFS, PR, and SSSP. In SSSP, for instance, the vector containing is updated iteratively by multiplying a row of the matrix by the path-length vector and then choosing the *minimum* of the result vector. After traversing all the nodes, the final values of the vector indicate the shortest paths from a source node to all the other nodes. PR iteratively updates the rank vector, initialized by equal values. At

each iteration, the elements of the rank vector are divided by the elements of the out-degree vector (i.e., the number of out-going edges for each vertex), chosen by a row of the matrix, and the result vector is reduced to a single rank by *adding* the elements of the vector.

Common features: While the sparse kernels used in both scientific and graph applications are similar in having sparse matrix operands, some kernels (e.g., SpMV) exhibit more concurrency, whereas others (e.g., SymGS) have several data dependencies in their computations. Regardless of this difference, a common property of kernels is that the reuse distance of accesses to the sparse matrix is high, while the input and output vectors of these kernels are being reused frequently. Moreover, the accesses to at least one of the vectors are often irregular. The other and more important, common feature of these kernels is that they follow the three phases of operations iteratively as listed in Table 4.1. The sparse kernels calculate an element of their result by accessing a row/column of the sparse large matrix only once and then reuse one or two vector/s for the calculation of all output vector elements. We benefit from the common features to generalize our proposed hardware without significant overhead. Alrescha converts the sparse kernels into the dense data paths, listed in the third column of Table 4.1 (details in the following).

Table 4.1: The properties of sparse kernels and corresponding dense data paths, implemented in Alrescha. Depending on the type of kernel, the *operation* in phase 1 can use the three vector operands at the same time or use just two of them.

Sparse Kernel	Sparse Application	Dense Data Paths	Phase 1 (vector operation)				Phase 2 (reduce)	Phase 3 (assign)
			vector operand1	vector operand2	vector operand3	operation		
SymGS	PDE solving	D-SymGS/GEMV	a row of coefficient matrix	the vector from iteration (i-1)	the vector at iteration (i)	multiplication	sum	apply operation with A^T and b_j and update vector
SpMV	PDE solving and graph	GEMV	a row of coefficient matrix	the vector from iteration (i-1)	N/A	multiplication	sum	sum and update the vector
Page Rank	Graph	D-PR	a column of adjacency matrix	the out-degree vector of vertices	the rank vector at iteration (i-1)	AND/division	sum	rank vector update
BFS	Graph	D-BFS	a column of adjacency matrix	the frontier vector	N/A	sum	min	compare and update distance vector
SSSP	Graph	D-SSSP	a column of adjacency matrix	the frontier vector	N/A	sum	min	compare and update distance vector

4.4 Putting Them Together for Sparse PDEs

Alrescha is a memory-mapped accelerator, the memory of which is accessible by a host for programming. Figure 4.4 shows an overview of Alrescha, the host, and the connections for programming and transferring data. The programming model of Alrescha is similar to offloading computations from a CPU to a GPU. To program the accelerator, the host launches the sparse kernels of sparse algorithms (e.g., PCG) to the accelerator. To do so, the host first converts the sparse kernels into a sequence of *dense data paths* and generates a binary file. Then, the host writes the binary file to a configuration table of the accelerator through the *program interface*.

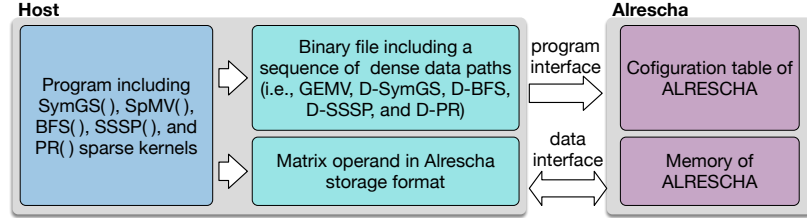


Figure 4.4: The overview of Alrescha and host.

During the execution of an algorithm, repetitive switching between the dense data paths is required. The key feature of Alrescha to enable fast switching among those dense data paths is the real-time partial reconfigurability. The details of the reconfigurable microarchitecture of Alrescha and the mechanism of real-time reconfiguration are explained in Figure 4.5. Besides switching among the data paths during runtime, Alrescha also reorders the dense data paths to reduce the number of switches. Such a reordering necessitates the new compression format, introduced in section 4.2. Therefore, the other task of the host is to reformat the sparse matrix operands into the compression format consisting of blocks, each of which corresponds to a dense data path. The formatted data is written into the physical memory space of the accelerator through the *data interface* (Figure 4.4).

Since the target algorithms are iterative, the preprocessing (i.e., conversion and reformatting) is a one-time overhead. Besides, the complexity and effort of preprocessing de-

pend on the previous format, data source, and host platform. For instance, the conversion complexity from frequently-used storage formats (e.g., CSR and BCSR) is linear in time and requires constant space. Since the preprocessing complexity is linear, it can be done while data streams from the memory. Moreover, if data is generated in the system (e.g., sensors), it is initially be formatted in the Alrescha format and reformatting is not required.

Algorithm 1 Convert Algorithm

```

1: function CONVERT( $A_{n \times n}, \omega$ , KernelType)
    $A_{n \times n}$ : sparse matrix,  $\omega$ : block width
    $DP$ : Data path type
    $l2r$ : left to right,  $r2l$ : right to left
2:    $Inx_{in} := 0, Inx_{out} := 0$ 
3:    $Blocks[] = \text{Split}(A, \omega)$  // partitions A to  $\omega \times \omega$  blocks
4:    $m = n/\omega$ 
5:   for ( $i = 1, i < m, i++$ ) do
6:     for ( $j = 1, j < m, j++$ ) do
7:       if ( $\text{nnz}(Blocks[i, j]) > 0$ ) then
8:         if KernelType  $\neq$  SymGS then
9:            $DP = \text{KernelType.DataPath}$ 
10:           $Inx_{in} = i.\omega, Inx_{out} = j.\omega$ 
11:           $Order = l2r$ 
12:           $Op = port1$  // the operand vector
13:        else
14:          if ( $i \neq j$ ) then
15:             $DP = \text{GEMV}$ 
16:             $Inx_{in} = j.\omega, Inx_{out} = -1$ 
17:             $Order = l2r$ 
18:            if ( $i > j$ ) then
19:               $Op = port2$  //which is  $x^{t-1}$ 
20:            else
21:               $Op = port1$  //which is  $x^t$ 
22:            end if
23:          else
24:             $DP = \text{D-SymGS}$ 
25:             $Inx_{in} = j.\omega, Inx_{out} = (i + 1).\omega$ 
26:             $Order = r2l$ 
27:             $Op = port2$  //which is  $x^{t-1}$ 
28:          end if
29:          end if
30:           $\text{Add2Table}(DP, Op, Inx_{in}, Order, Inx_{out})$ 
31:        end if
32:      end for
33:    end for
34:  end function

```

Algorithm 1 shows the procedure for converting a sparse matrix to dense data paths. The general procedure of the conversion algorithm is as follows: (i) As lines 8 to 12 show, the sparse kernels with no (or straightforward) data dependencies including SpMV, BFS, SSSP, and PR are broken down into a sequence of general matrix-vector multiplication (GEMV), dense BFS (D-BFS), dense SSSP (D-SSSPs), and dense PR (D-PR), respectively. These dense data paths have the same functionality as their corresponding sparse kernels do; however, they work on *non-overlapping locally-dense blocks* of the sparse ma-

trix operand and *overlapping sub-vectors* of the dense vector operand of the original sparse kernel. (ii) As lines 13 to 26 show, the sparse kernels with data dependencies (e.g., SymGS kernel) are broken down into a majority of parallelizable GEMV (lines 14 to 21) and a minority of sequential dense SymGS (D-SymGS) data paths (lines 23 to 26).

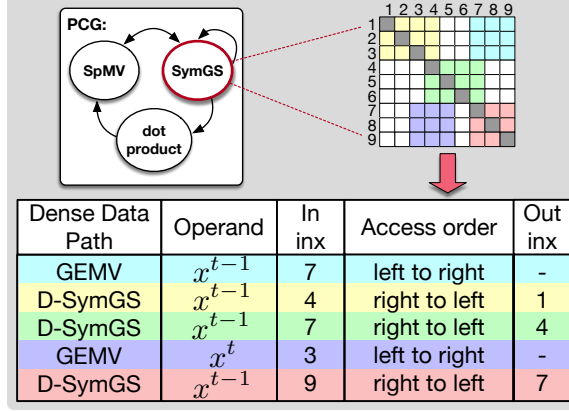


Figure 4.5: The order of operations: An example of the configuration table for a SymGS kernel, in which $n = 9$, $\omega = 3$.

The conversion for SymGS is to assign GEMVs to non-diagonal non-zero blocks (line 15) and D-SymGS to diagonal non-zero blocks of the sparse matrix (line 23). For accelerating SymGS, the key insight of Alrescha is to separate GEMV from D-SymGS data paths to prevent the performance from being limited by the sequential nature of the SymGS kernel. To this end, Alrescha reduces switching between GEMV and D-SymGS by reordering them so that Alrescha first executes all the GEMVs in a row successively and then switches to a D-SymGS. The distributive property of inner products in

$$x_j^t = \frac{1}{A_{jj}^T} - (b_j - \sum_{i=1}^{j-1} A_{ij}^T \times x_i^t - \sum_{i=j+1}^n A_{ij}^T \times x_i^{t-1}). \quad (4.1)$$

guarantees the correctness of such reordering. As an example of the outcome of Algorithm 1, Figure 4.5 shows the state machine of PCG, which comprises three sparse kernels, two of which are the focus of this paper and are launched to the accelerator by the host. The configuration table for a SymGS example is shown in Figure Figure 4.5. Based on Equation Equation 4.1 and as lines 19 and 21 of Algorithm 1 indicate, all the non-zero

blocks in the upper triangle of A have to be multiplied by x^t , and all of those in the lower triangle have to be multiplied by x^{t-1} .

4.5 Reconfigurable Microarchitecture

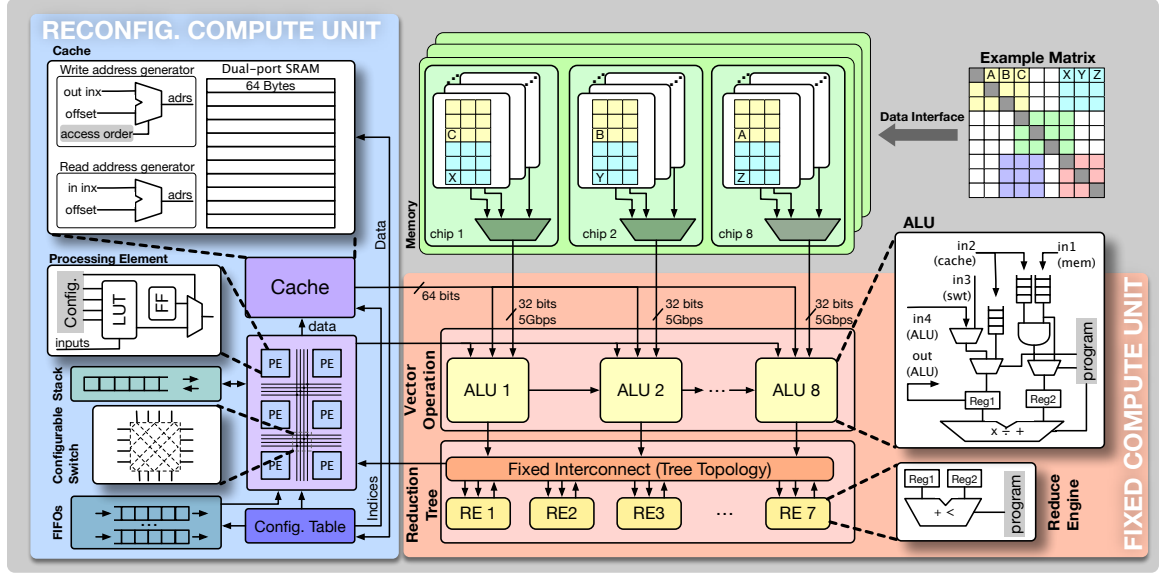


Figure 4.6: The microarchitecture of Alrescha: (a) the FCU for implementing common computations, and the RCU for providing specific configuration for distinct dense data paths. Example configurations for supporting: (b) D-SymGS, (c) GEMV, and (d) D-PR.

Here, we introduce the microarchitecture of Alrescha. The key feature of the proposed microarchitecture is partial reconfigurability. The benefit of this feature is two-fold. First, for SymGS, we have seen that the building blocks of GEMV and SymGS require a common core hardware mechanism. Besides, the GEMVs create a big portion of operations. Therefore, as long as Alrescha is performing subsequential GEMVs, it does not have to select what to do neither by decoding an instruction nor by selecting a path in the hardware. The second benefit of partial reconfiguration goes to the other applications (e.g., graph kernels) that also share a core hardware mechanism. As a result, Alrescha can simply perform other applications (or one application including distinct kernels) without needing to change the entire hardware. Reconfiguring only a fraction of the entire data path reduces

the configuration time. To achieve the goal of partial reconfiguration, Alrescha consists of a separate fixed computation unit (FCU) and a reconfigurable computation unit (RCU) and configuring only the former for switching between data paths (Figure 4.6).

4.6 Evaluation

4.6.1 Experimental Setup

This section explores the performance of Alrescha by comparing it with the CPU, GPU, and state-of-the-art sparse accelerators. We evaluate Alrescha for both scientific applications and graph analytics. We pick real-world matrices with applications in scientific and graph problems from the SuiteSparse Matrix Collection [117]. We run PCG, which includes the SymGS and SpMV kernels, on the matrices with a scientific application, and run graph algorithms (i.e., BFS, SSSP, and PR) on the graph matrices. We also run SpMV on both categories of datasets.

We compare Alrescha with the CPU and GPU platforms. The configurations of the baseline platforms are listed in Table 4.2. For the CPU and GPU, we exclude disk access time. For fair comparisons, we include optimizations, such as row reordering and suitable storage formats (e.g. ELL) proposed for the CPU and GPU implementations. The PCG algorithm and the graph algorithms running on GPU are respectively based on the cuSPARSE and Gunrock [120] libraries. The graph algorithms running on the CPU are based on the GridGraph [121] and/or CuSha [122] platforms (whichever achieves better performance). Besides the comparison with the CPU and GPU, this section compares Alrescha with the state-of-the-art hardware accelerators, including OuterSPACE [69], an accelerator for SpMV, GraphR [72], a ReRAM-based graph accelerator, and a Memristive accelerator for scientific problems [73]. To reproduce their latency and power consumption numbers, we modeled the behavior of the preceding accelerators based on the information provided in the published papers (e.g., the latency of read and write operations for GraphR and Memristive accelerator). We validate our numbers based on their reported numbers for their

configurations to make sure our reproduced numbers are never worse than their reported numbers. Seeking a fair comparison, we assign all the accelerators the same computation and memory-bandwidth – this assumption does not harm the performance of our peers.

Table 4.2: Baseline configurations.

GPU baseline	
Graphics card	NVIDIA Tesla K40c, 2880 CUDA cores
Architecture	Kepler
Clock frequency	745MHz
Memory	12 GB GDDR5, 288 GB/s
Libraries	Gunrock [120] and CUSPARSE
Optimizations	row reordering (coloring) [77], ELL format
CPU baseline	
Processor	Intel Xeon E5-2630 v3 8-core
Clock frequency	2.4 GHz
Cache	64 KB L1, 256 KB L2, 20 MB L3
Memory	128 GB DDR4, 59 GB/s
Platforms	CuSha [122], GridGraph [121]

We convert the raw matrices using Algorithm 1 implemented in Matlab. To do that, we examine block sizes of 8, 16, and 32 for the range of data sets and choose the block size of eight because, unlike the other two, 8 provides a balance between the opportunity for parallelism and the number of non-zero values. We model the hardware of Alrescha using a cycle-level simulator with the configurations listed in Table 4.3. The clock frequency is chosen to enable the compute logic to follow the speed of streaming from memory (i.e., each 64-bit operands of ALU are delivered from memory in 0.4 ns, through the 32-bit 5 Gbps links.) To measure energy consumption, we model all the components of the microarchitecture using a TSMC 28 nm standard cell and the SRAM library at 200 MHz. The reported numbers include programming the accelerator.

While FPGAs have had the partial reconfiguration feature for over a decade, fewer ap-

Table 4.3: Alrescha Configuration.

Floating point	double precision (64 bits)
Clock frequency	2.5 GHz
Cache	1KB, 64-Byte lines, 4-cycle access latency
RE latency	3 Cycles (sum: 3, min: 1)
ALU latency	3 Cycles
Memory	12 GB GDDR5, 288 GB/s

plications have been proposed to use it. Alrescha proposes a new application for partial reconfiguration. Our goal is to leverage partial reconfigurability to evaluate the switching between the different algorithms, without fully reprogramming the FPGA hence we utilize *static* partial reconfiguration rather than a *dynamic* one – note that dynamic reconfiguration can also be implemented for SymGS. We implement SymGS and graph algorithms, the common function of which is a matrix-vector multiplication. We implement Alrescha using Xilinx Vivado HLS. We use relevant *#pragma* as hints to describe our desired microarchitectures in C++. We target Xilinx AC701 evaluation kit, including a partially reconfigurable Artix-7 FPGA, XC7A200T. We present the post-implementation resource utilization and power consumption, reported by Vivado. Inputs and outputs of Alrescha are transferred through the AXI stream interface. The clock frequency is set to 200 MHz.

4.6.2 Execution Time

Scientific Problems: The primary axis of Figure 4.7 (i.e., the bars) illustrates the speedup of running PCG on Alrescha over the GPU implementation optimized by row reordering [77] for extracting a high level of parallelism; the secondary axis of Figure 4.7 shows the bandwidth utilization. The figure also captures the speedup of the Memristive-based hardware accelerator [73]. On average, Alrescha provides a $15.6\times$ speedup compared to the optimized implementation on the GPU. The speedup of Alrescha is approximately twice that

of the most recent accelerator for solving PDEs. To investigate the reasons behind this observation, we plot memory bandwidth utilization in Figure 4.7. As the figure shows, the performance of Alrescha and the other hardware accelerator for all scientific datasets is directly related to memory bandwidth utilization – mainly because of the sparsity nature. Moreover, none of them fully utilize the available memory bandwidth because both approaches use blocked storage formats, in which the percentage of non-zero values in a block rarely reaches a hundred percent. Nevertheless, we see that Alrescha better utilizes the bandwidth because it resolves the dependencies in computations, which otherwise limits bandwidth utilization.

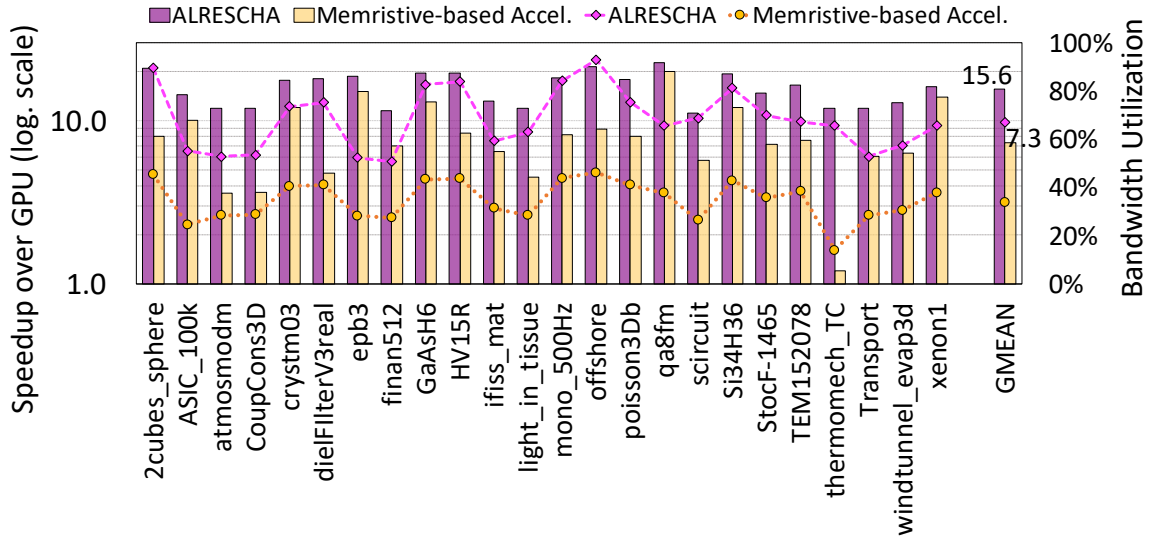


Figure 4.7: Speedup: PCG algorithm on scientific datasets, normalized to GPU (bar charts), and bandwidth utilization (the lines) compared to the state-of-the-art accelerator for scientific problems [73].

To clarify the impact of resolving dependencies on overall performance, Figure 4.8 presents the percentage of data-dependent computations in the GPU implementation, versus that in Alrescha, which has an average of 23.1% data-dependent operations. As the figure suggests, even in the GPU implementation that extracts the independent parallel operations using row reordering and graph coloring, on average 60.9% of operations are still data-dependent. This is more than 60% for highly-diagonal matrices and less than 60% for matrices with a greater opportunity for in-row parallelism. Such a trend identifies

the distribution of locally-dense blocks as another rationale for determining the speedups. More specifically, when the distribution of non-zero values in rows of a matrix offers the opportunity for parallelism, the speedup over the GPU is smaller than when the matrix is diagonal. Therefore, to conclude, for multi-kernel sparse algorithms with data-dependent computations, Alrescha improves performance by (i) extracting parallelizable data paths, (ii) reordering them and the elements in the blocks to maximize the reuse of data, and (iii) implementing them in lightweight reconfigurable hardware, which results in fast switching not only between the distinct data paths of a single kernel but also among them.

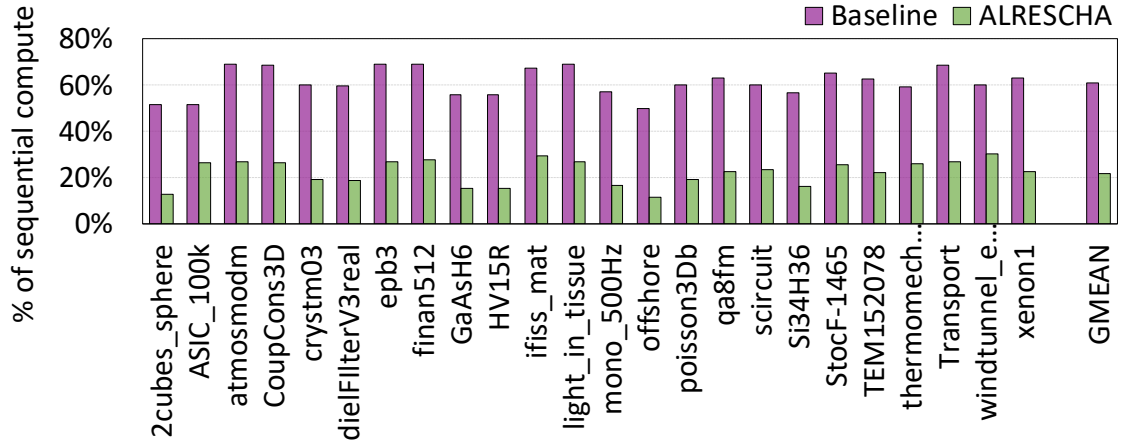


Figure 4.8: Reduction in data-dependent operations: for the PCG algorithm, after applying Alrescha. The baseline shows the percentage of data-dependent operations by row-reordering optimization.

Graph Analytics & SpMV: This section explores the performance of the algorithms consisting of a single type of kernel with fewer data dependency patterns in their computations. Such a study claims that Alrescha is not just optimized for a specific domain and is applicable to accelerating a wide range of sparse applications. First, we analyze the performance of graph applications. Figure 4.9 illustrates the speedup of running BFS, SSSP, and PR on Alrescha, a recent hardware accelerator for graph applications (i.e., based on GraphR [72]), and GPU, all normalized to the CPU. As the figure shows, Alrescha offers average speedups of $15.7\times$, $7.7\times$, and $27.6\times$, for BFS, SSSP, and PR algorithms, respectively. We achieve this speedup by avoiding the transfer of meta-data, reordering the blocks

for increasing data reuse and improving the locality. Further, to run graph applications, Alrescha performs only subsequential same-type dense data paths that eliminates the need to neither decode instructions nor select a data path in the hardware.

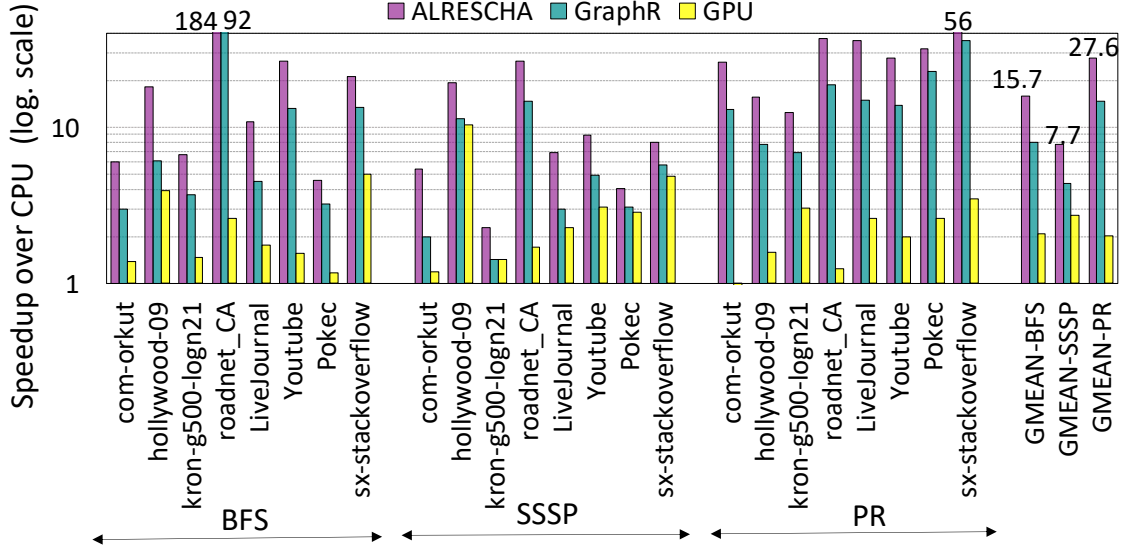


Figure 4.9: Speedup: graph algorithms on graph datasets over the CPU. GraphR [72] is the state-of-the-art graph accelerator.

The primary axis of Figure 4.10 (i.e., the bars) illustrates the speedup of SpMV, a common algorithm of various sparse applications on Alrescha and OuterSPACE [69] (i.e., the recent hardware accelerator for the SpMV), normalized to the GPU baseline. As the figure shows, Alrescha offers average speedups of $6.9\times$ and $13.6\times$ for scientific and graph datasets. When running SpMV, all the data paths are GEMV; therefore, no transmission between data paths is required. However, optimizations of Alrescha help achieve greater performance. The key optimization here is accesses to the cache to obtain frequent accesses to the vector operand of SpMV. To show this, the secondary axis of Figure 4.10 (i.e., the lines) plots the percentage of the whole execution time for accesses to the local cache. Alrescha utilizes locality in cache accesses (i.e., consuming the values in a cache line in succeeding cycles), and increases the data reuse rate of not only the input sparse-matrix operands but also the dense-vector operands and output vector. Although in the outer-product approach, data read from the cache is broadcast to all the ALUs, to be reused

as many times as required, before being written back to the cache, an element of the output vector must be fetched several times. During such accesses to the cache, the spatial locality of non-zeros is not captured. On contrary, the approach of Alrescha that applies GEMV to locally dense blocks of the sparse matrix instead of working on individual non-zeros takes advantage of spatial locality in the non-zero values of the sparse matrix. Besides, Alrescha sums up the results of multiplications locally, without redundant accesses to the cache. To do so, Alrescha splits the vector operand into chunks and at each time step, instead of fetching an individual element, it fetches a chunk of vector operand from the cache, and instead of broadcasting, it sends them to individual ALUs. The elements of a chunk are multiplied by all the non-zero blocks of the sparse matrix in a row. As a result, each element of the output vector is fetched from cache only once per $\#cols/n$ (n : chunk size).

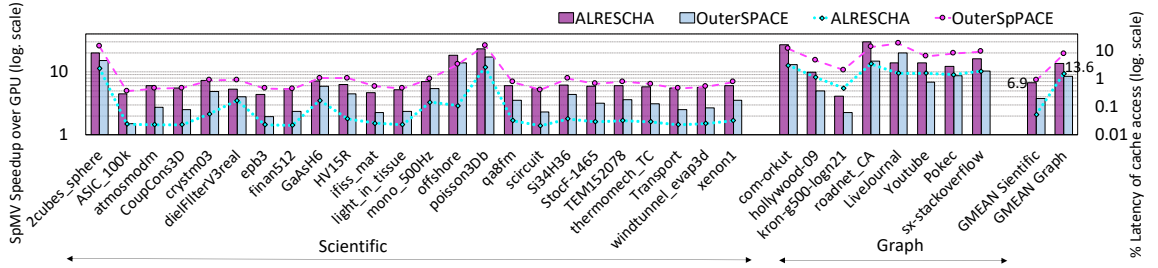


Figure 4.10: Speedup: executing SpMV on scientific and graph datasets normalized to GPU (bar charts), and the percentage of execution time devoted to cache accesses (the lines). OuterSPACE [69] is the state-of-the-art SpMV accelerator.

4.6.3 Energy Consumption

To improve energy consumption, the techniques integrated into the hardware accelerators have to be efficient. A source of energy consumption is accessing local SRAM-based buffers or caches. That is, reducing the number of reads and writes from and to local memories, by substituting them with computation is beneficial. Figure 4.11 illustrates the energy consumption of Alrescha for running SpMV, normalized to that of the CPU and GPU baselines. As Figure 4.11 shows, on average, the total energy consumption improves by $74\times$ compared to the CPU and $14\times$ compared to the GPU platform. Note that the

activity of computing units, defined by the density of the locally-dense block, impacts energy but not performance. In summary, the main reasons for the low energy consumption are the small reconfigurable hardware of Alrescha in combination with utilizing a storage format with the right order of blocks and values matched with the order of computation to avoiding the decoding the meta-data and reducing the number of accesses to the memory.

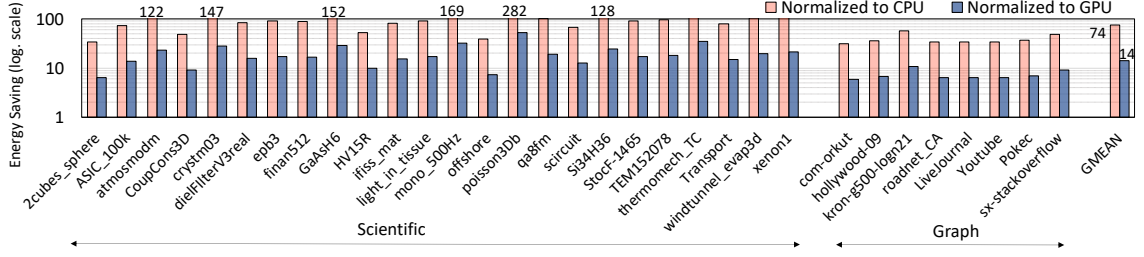


Figure 4.11: Energy consumption of Alrescha normalized to CPU and GPU.

4.6.4 FPGA Resource Utilization and Power Consumption

Here, we evaluate the implementation of Alrescha on our target partially reconfigurable FPGA (i.e., Artix-7 FPGA, XC7A200T) consisting of total 433K LUTs and 174K FFs), the features of which align with one another. Table 4.4 compares the resource utilization and dynamic power consumption of a static design with a partially-reconfigurable design including FCU and RCU. As the table suggests, in the static design, each of the dense data paths utilizes the resource as much as required. Therefore, D-PR, which includes division operations, utilizes the most number of LUTs and FFs and the highest power consumption, whereas D-BFS, which requires the simplest design, utilizes the minimum resources. On the other hand, since in the partially reconfigurable design, the FCU must envision the verity of operations, the overall architecture utilizes more than enough resources for GEMV, D-BFS, and D-SymGS. The RCU, however, is tailored to each design. Thus, D-SymGS, with the most complex RCU utilized more resources than other data paths do. Although in some case, the partially reconfigurable design utilized more resources, the overheads are outweighed by its benefits, especially for multi-kernel applications such as the PCG algorithm that requires switching between distinct kernels during the run time. As the table

suggests, the partial reconfigurable implementation of the simple single-kernel workloads (i.e., SpMV and BFS) has an overhead of $2.9\times$ more LUT and $1.9\times$ more FF as well as $1.2\times$ more dynamic power consumption compared to the static implementation. On the contrary, for more complex single-kernel workloads such as PR and the multi-kernel workloads such as PCG (including SymGS and GEMV), the partially reconfigurable implementation is more beneficial as it utilizes $1.2\times$ fewer LUTs and FFs, and consumes $1.15\times$ less dynamic power consumption. Note that here we show only the core computation unit. However, the fixed modules involve many other components of a complete architecture.

Table 4.4: Resource utilization and the total dynamic power consumption.

	Static Design				
	GEMV	D-PR	D-BFS	D-SymGS	
LUT	2386	8632	2246	3467	
FF	6489	10233	4439	7845	
Power(W)	0.098	0.115	0.065	0.102	
	Partially Reconfigurable Design				
	FCU	RCU			
		GEMV	D-PR	D-BFS	D-SymGS
LUT	6594	271	271	123	645
FF	9771	380	380	320	1594
Power(W)	0.086	0.03	0.03	0.01	0.06

4.7 Summary

This chapter showed that sparse scientific computing that dominate a wide range of applications fail to effectively benefit from high memory bandwidth and concurrent computations in modern high-performance computer systems. Therefore, hardware accelerators have been proposed to capture a high degree of parallelism in sparse problems. However, the unexplored challenge for scientific computing is the limited opportunity for parallelism because of data dependencies. The key insight proposed in this chapter is to extract parallelism by mathematically transforming the computations into equivalent forms. The

transformation breaks down the sparse kernels into a majority of independent parts and a minority of data-dependent ones and reorders these parts to gain performance. To implement the key insight, this chapter proposed Alrescha, a lightweight reconfigurable sparse-computation accelerator. To efficiently run the data-dependent and parallel parts and to enable fast switching between them, Alrescha made two contributions. First, it implements a compute engine with a fixed compute unit for the parallel parts and a lightweight reconfigurable engine for the execution of the data-dependent parts. Second, Alrescha benefits from a locally-dense storage format, with the right order of non-zero values to yield the order of computations dictated by the transformation. The combination of the lightweight reconfigurable hardware and the storage format enables uninterrupted streaming from memory. Our simulation results showed that compared to GPU, Alrescha achieves an average speedup of $15.6\times$ for scientific sparse problems, and $8\times$ for graph algorithms. Moreover, compared to GPU, Alrescha consumes $14\times$ less energy.

CHAPTER 5

STRUCTURED PRUNING FOR SYSTOLIC ARRAYS

This chapter focuses on the inference of neural networks, the third category of sparse problems that are studied in this dissertation. Unlike the last two chapters that made changes in hardware to make software optimizations to be effective and impactful, this chapter proposed some modifications in *software* and the way we represent it to better utilize an efficient *hardware*. To resolve the challenge of computing underutilization of systolic arrays for the inference of CNNs, we propose creating locally-dense DNNs for efficient inference on systolic arrays (Lodestar*) [5]. Lodestar produces weight matrices such that the non-zero values are clustered spatially into *locally-dense* regions, which are compactly stored and efficiently *streamed* from memory. To efficiently run inference of DNNs using systolic arrays, we propose Eridanus[†] [19], a systolic-based compute engine with a streaming interface. Eridanus efficiently handles indexing and uses the execution model of Mahasim[‡] [123] (machine-learning hardware acceleration using a software-defined intelligent memory system).

5.1 Lodestar Pruning Algorithm

Lodestar is a pruning algorithm consisting of the following key insights: (i) To capture the data reuse patterns in systolic arrays and enable data streaming, modifying the *distribution* of non-zero values is more influential than minimizing the number of operations or the memory footprint; and (ii) To achieve an appropriate distribution of non-zero values, examining the correlation among the filters rather than the individual filters increases the chances of creating a systolic-friendly model. Therefore, in contrast to other pruning algo-

*a star that is used to guide the course of a ship, especially the Pole Star.

[†]a constellation in the southern hemisphere.

[‡]a binary star in the constellation of Auriga

rithms, instead of the individual filters, Lodestar examines and prunes the flattened weight matrix by *extracting the potential non-zero blocks, the widths of which are matched the width of the target systolic array*. The blocks are created by hypothetically splitting the weight matrix into F^2C/ω chunks (F^2C : the common axis of input and weight matrix, ω : the width of the systolic array), and then, extracting the non-zero blocks in each chunk. While the widths of the blocks must match the widths of the systolic arrays to guarantee the correctness of multiplications, their length could be arbitrary. To reduce the complexity of the algorithm, we choose a fixed length. Once the blocks are extracted, the adjacent ones are concatenated and stored as a single block by assigning it a single index (i.e., the column index of the first block) and a single length.

Algorithm 2 illustrates the Lodestar pruning algorithm, the input parameters of which are the weight matrix W , threshold θ , the length of the window (l , a hyperparameter), and the width of the systolic array (ω). The width of the window is fixed and is equal to ω . The weight matrix is either the flattened version of the weight matrix in a convolution layer or the 2D weight matrix itself in a fully-connected layer. During pruning, a window of size $\omega \times l$ slides over W . If the average value of the window is smaller than θ , the block corresponding to that window is set to zero. During retraining, the threshold (θ) of the average values for choosing/pruning the zero blocks is gradually increased with training epochs. The windows are non-overlapping in x- and y-axes. The non-overlapping window in x-axes is necessary to match with systolic-array width, and in y-axes for reducing the complexity of the problem from a global to local optimization.

Algorithm 2 does not change the size of the common axis of the operands of GEMM (i.e., F^2C), which leads to following benefits: (i) no need to change the dimensions of the image, and (ii) both the pruned matrix (i.e., weights) and the dense matrix (i.e., inputs) can be either streamed through the systolic array or be the stationary operand during the multiplication (details in section 5.2). Thus, based on the size of the matrices at each layer, we can dynamically swap the role of the two matrices to be streamed or stationary.

Algorithm 2 Pruning

```

1: function PRUNE( $W_{h \times w}, \theta, l, \omega$ )
    $W_{h \times w}$ : Weight matrix,  $\theta$ : Threshold,
    $\omega$ : Systolic array width  $l$ : Window length
2:    $i_h := 0, i_w := 0, avg := 0$ 
3:   while  $i_w < w$  do
4:      $avg = \text{BlockAvg}([i_w, i_h], [i_w + \omega - 1, i_h + l - 1])$ 
5:     if  $avg < \theta$  then
6:        $W[i_w : i_w + \omega - 1, i_h : i_h + l - 1] = 0$ 
7:        $i_h = i_h + l$ 
8:     else
9:        $i_h = i_h + 1$ 
10:    end if
11:    if  $i_h > h - l$  then
12:       $i_h = 0$ 
13:       $i_w = i_w + \omega$ 
14:    end if
15:  end while
16: end function

```

5.2 Eridanus Systolic Microarchitecture

Eridanus uses a weight-stationary systolic array with one streaming and one stationary input (i.e., $R1$ s and $R2$ s in Figure 5.1). The streaming input registers are connected in a column and their contents shift one row down ❶, at each cycle. The stationary registers are also connected to simplify the interconnection between the array and memory. The input from memory is connected to the first row. The stationary data swing through registers until they reaching the destination. The streaming registers and the stationary registers share memory bandwidth to obtain their contents. The outputs of a row are summed through an adder tree to contribute to creating an element of the output. The number of adder trees defines the number of output elements generated at each cycle.

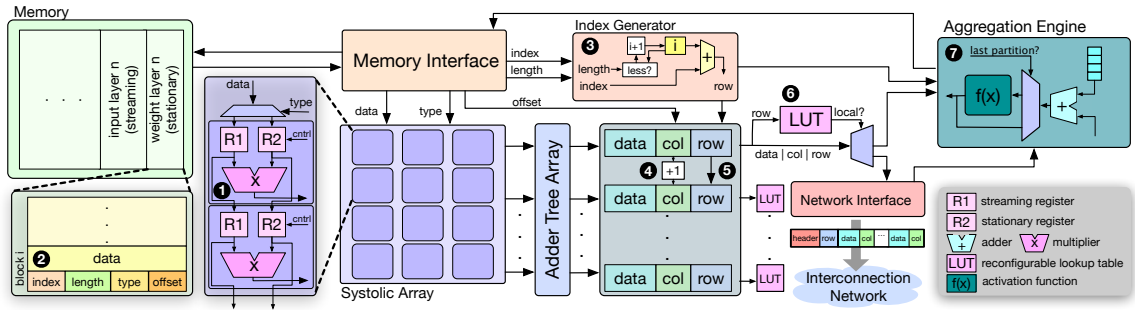


Figure 5.1: An overview of the systolic-based microarchitecture of Eridanus.

Since the width of the systolic array defines the *degrees of concurrency*, we want it to

match the width of the 2D-weight matrix to maximize the fine-grained parallelism. However, to be flexible and scalable, we prefer to employ several narrow systolic arrays, instead of one large array, so that based on the size of a weight matrix, we assign as many narrow systolic arrays as required. On the other hand, the depth of the systolic array directly impacts the data-reuse rate. Thus, a deeper systolic array is often preferred. However, pruning causes variations in the length of the locally-dense blocks, which are going to stay in the systolic array. Therefore, while choosing a large depth for the systolic array leads to under-utilization of the systolic array, a very small depth prevents achieving peak throughput. To optimize for the common case, we choose a depth of 64.

To maximize bandwidth utilization and avoid random memory accesses, we map locally-dense weights and the inputs corresponding to sequential multiplications in the sequential addresses. Therefore, for each layer, the stationary operand is streamed, followed by the streaming operand. The *type* identifier, which indicates stationary and streaming data, is used to direct data to the registers. The headers of blocks include the index, the length, type of data, and an offset ❷. When the width of the stationary operand matrix is larger than the depth of the systolic array, we split a multiplication into sub-multiplications. The *offset* is the index of the sub-multiplications and is used to generate the column-index of the output elements. The memory interface reads from memory and directs *data* and *type* to the systolic array, and sends the *index*, *length*, and *offset* of blocks to the index generator ❸.

In multiplying $W_{K \times F^2C} \times I_{F^2C \times WH}$, the column and row indices of the output elements are defined by the column index of I and the row index of W , respectively. Therefore, the position of an adder tree simply indicates the column index of the output elements – the offset is added to it if the matrix does not fit in the systolic array. This is implemented by the increment units between the column indices ❹. The block index and length indicate the row within the selected column of the output. As the row indices of the output elements corresponding to a single block are sequential, they are reused by shifting them down ❺.

When $F^2C > \omega$, more than one systolic array will contribute to calculating an ele-

ment of the output. As a result, the partial results will have to be aggregated in the final destination (i.e., based on the mapping of the next layer across the systolic arrays). The mapping of the sub-multiplications to the systolic arrays is programmed in a look-up table (LUT) ⑥. Once the column and row indices are assigned to the outputs of the adder trees, based on the LUT, they will be directed to other systolic arrays or the aggregation engine of the current one. The aggregation engine ⑦ sums the partial results, and if the current partial result is the last portion of the output element, it applies the activation function to the final result and sends it to the memory interface to be written in memory. If an element of output, it is directed to the network interface. We use a multi-drop express channels (MECS) topology, a bandwidth-efficient interconnection network. The elements with the same destination, which also have a common row index, are packetized together. When a packet is received, it is de-packetized and sent to the aggregation engine.

5.3 Evaluation

5.3.1 Experimental Setup

We use Tensorflow to apply Lodestar on VGG16, CifarNet, and LeNet, Cifar10. Our baselines are pruning the same models using state-of-the-art pruning algorithms [92, 124, 91, 89, 93, 90] and running them on a systolic-based engine (when required, additional buffering/caching mechanisms are implemented). We prune all the models to achieve equal accuracy and hence various sparsity. Seeking fair comparison, each baseline pruning method is compiled based on its best format. The pruned models are used as the input to our in-house cycle-level simulator that models the microarchitecture. We use High Bandwidth Memory (HBM) as the memory connected to 8×64 systolic arrays. We estimate the power consumption of the compute units by using Kitfox1.1 library at 16nm technology and McPAT model. We assume the access energy per bit of 6 pJ/bit for HBM. We connect eight modules shown in a MECS topology, in which a packet consumes 0.52 nJ energy at routers and links. The latency of each multiplier is three cycles @2GHz. We process batches of size

16. By choosing a relatively small batch size, we neither increase the number of reloads at mid-size layers nor destroy the compute utilization at fully-connected layers.

5.3.2 Accuracy

Figure 5.2a illustrates the top-1 accuracy of the CifarNet and VGG16 pruned by Eridanus, normalized to the accuracy of the unpruned model, along with the average percentage of zero blocks. For CifarNet (Figure 4a), pruning is applied between steps 20k and 100k. For VGG16, since we use a pre-trained model, we start pruning from the beginning (i.e., step 1 to 10k). As Figure 5.2a shows, during pruning, the percentage of zero blocks increases. However, since the distribution of zero blocks and/or their densities keep changing, the accuracy oscillates. After pruning stops, training continues to maximize the accuracy by adjusting the values of non-zero blocks. For LeNet, CifarNet, and VGG16, we prune 75%, 79.8%, and 42% of models and respectively achieve 99%, 93.6%, and 70% top-1 accuracy on the validation set. The top-1 accuracy of unpruned models is 99% for LeNet, 94% for CifarNet, and 71.5% for VGG-16. Note that parameters such as threshold (θ in Algorithm 2), the start and the end steps, the length of the sliding window, and the maximum desirable sparsity impact the trade-off between accuracy and zero distribution.

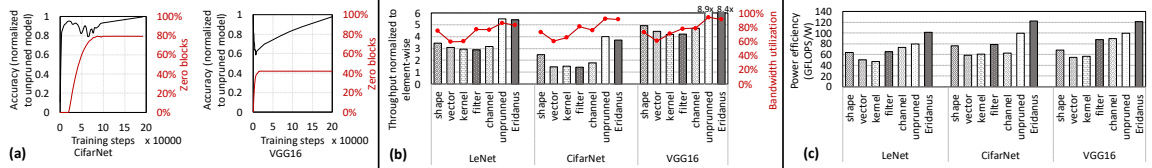


Figure 5.2: The results of implementing Lodestar on Eridanus: (a) The accuracy and the percentage of zero blocks for CifarNet (pruned between step 20k, 100k) and VGG16 (pruned between steps 1, 10k). (b) Throughput and bandwidth utilization. (c) the power efficiency of three DNN models pruned by various structured techniques.

5.3.3 Performance

Throughput & bandwidth utilization: The level of concurrency, the number of times we need to load the stationary operands, the additional caching/buffering/decoding for direct-ing data from memory into the systolic array, and memory bandwidth utilization impact

throughput. Bandwidth utilization depends on the number of indirect memory accesses and the density of the models. As a result, the unpruned model is expected to have the highest throughput and bandwidth utilization compared to structured models. Figure 5.2b shows the throughput and bandwidth utilization of DNN inference on systolic arrays for the models pruned with various granularities. The trend of the overall performance is similar to that of throughput since the amount of computation remains similar.

As Figure 5.2b illustrates, the bandwidth utilization and throughput of Eridanus is very similar to those of the unpruned models. The other approaches, however, are not as effective, because they are not jointly optimized for capturing the data-reuse patterns and concurrency. For instance, kernel-wise captures less data-reuse patterns, while it enables a high level of parallelism. On the other hand, compared to other baseline structured models, shape-wise and channel-wise yield better performance on systolic arrays, because they can capture more data-reuse patterns. However, they limit the level of concurrency. Although the number of operations for Lodestar could be more than those in irregular sparse models, their locality in Lodestar leads to lower latency. Thus, the combination of fast computation and high bandwidth utilization makes Eridanus closer to the peak throughput.

Power efficiency: The power consumption of inference on the systolic array is defined by the number of memory accesses as well as the number of operations. Comparing to other pruning approaches, Eridanus creates a model that requires the lowest number of memory accesses. However, the effect of pruning algorithms on the number of computations is the opposite. Comparing to element-wise pruning, the structured pruning approaches may have required a high number of operations. On the other hand, the systolic array executes spatial operations more quickly than sparse operations. As a result, the ratio of memory-access reduction to compute-density reduction is the key factor in defining the power efficiency. Figure 4c illustrates the combined effect of the number of memory accesses and computation density, on power efficiency. As the figure shows, for Eridanus, the reduction in memory accesses carries more weight and helps achieve higher power efficiency.

5.4 Summary

In this chapter, we saw that systolic arrays [40] have seen a resurgence for implementing the inference of CNNs, a practical example of which is in Google’s TPU [125]. We saw that since systolic arrays eliminate the need for irregular intermediate accesses to the memory hierarchy, they work particularly well for computing linear recurrences and *dense* linear algebra computations. However, the challenge presented in this chapter was that inference using CNNs is a *sparse* problem, which presents significant efficiency challenges such as underutilization of memory bandwidth due to storing data in sparse formats, indirect memory accesses and transferring of extra meta data.

CNN inference is sparse because, during training, several of weights are assigned close-to-zero values. Thus, to reduce computation and the memory footprint, the close-to-zero values are usually pruned. This chapter also showed that since pruning the individual values of a model results irregular models with consequences of resource underutilization and high storage overhead, *structured* pruning techniques have been proposed, which prune the weights at the granularity of a vector, kernel, filter, channel, or entire layer, all of which are optimizations for CPUs and GPUs and help in reducing the number of operations, memory footprint, and computation complexity. The main challenge is that the preceding optimizations are insufficient to exploit the data reuse in systolic arrays, and the highly concurrent, synchronous, and rhythmic flow of data from memory. In fact, the storage adjacency of data resulting from algorithm-defined pruning (e.g., kernel, filter) is not necessarily matched with data organizations necessary to directly stream to the interacting data flows in the systolic array.

This chapter introduced Lodestar and Eridanus to create locally-dense CNNs for efficient inference on *systolic arrays*, to enable streaming of sparse data from memory to exploit the distinctive *data reuse* patterns and fine-grained concurrency of systolic arrays. By using these techniques, we produce a *weight matrix* such that the non-zero values are

clustered spatially into *locally-dense* regions, which are compactly stored and efficiently *streamed*. We examine the correlation among all the filters, which differs from pruning the individual filters of a CNN. To sustain accuracy, we may keep more number of non-zeros compared to common pruning algorithms. As we showed in this chapter, in achieving higher performance and efficiency, the distribution of non-zeros is more influential than their quantity, when optimized for streaming data.

CHAPTER 6

FAST DECOMPRESSION

To see the challenges and opportunities for super sparse data such as those in scientific computing and graph analytics, this chapter explores the challenges associated with compression mechanisms that have been a common approach for sparse data. In other words, when data is too sparse, we compress it to better utilize storage. But when it is time to process, we need to do extra work to find out the original location of each nonzero to perform correct operation. Decompression is not necessarily fast and can potentially become a performance bottleneck when streaming data from memory into the computation units.

To achieve an ideal streaming accelerator for sparse problems, we propose Ascella *, the key insight of which is sustaining a balance between computing latency and data transfer rate. To do so, on one hand, Ascella avoids streaming the unnecessary *zero* elements to efficiently use the memory bandwidth; and, on the other hand, it provides fast decompression to keep following the speed of streaming. To enable the latter, Ascella *avoids extra accesses* to the buffers, and maintain *deterministic parallel accesses* to them, which are the two obstacles of using other well-known compressed storage formats (e.g., BCSR).

To avoid the overhead of extra accesses and enable deterministic parallel accesses to the buffers, we suggest using list-of-lists (LIL), a popular storage format supported by the SciPy library in Python. Figure 6.1 clarifies how using LIL reduces the number of cycles to read compressed data to decompress the non-zero rows. As Figure 6.1 shows, for each column of the original sparse matrix, LIL saves a list of row indices (i.e., *indices*) corresponding to each non-zero value; as well as a list of all non-zero elements in that column (i.e., *values*). The *columns* of values and indices (i.e., to blocks B0 to B7 in Figure 6.1) can always be accessed in parallel. As a result, no extra read access is required for determining

*a triple star system and the third brightest star in the constellation of Sagittarius.

the number of next read accesses. Creating a non-zero row takes the latency of one read access since they are parallel, plus the latency for creating the input of the dot product, which is smaller than those for decompressing CSR or BCSR because of simpler logic. Memory streaming time for Ascella is defined by the number of non-zero rows, the size of rows, and transferring one additional row for indicating the end of non-zero rows.

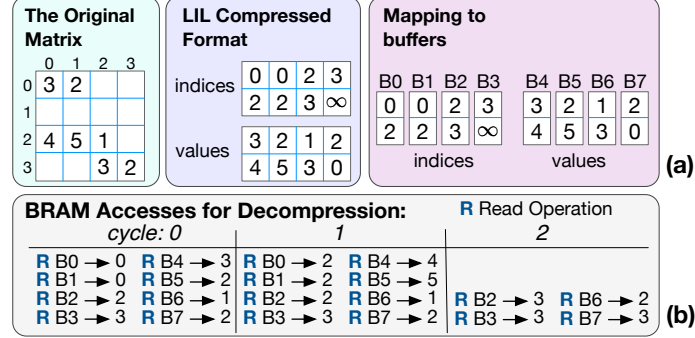


Figure 6.1: (a) Compressing a sparse matrix using LIL storage. (b) Time steps of reading indices and values to decompress the non-zero rows.

6.1 Ascella Decompression Mechanism

The components of our architecture to stream the compressed partitions and process them in an FPGA include (i) the global memory, (ii) an AXI streaming (AXIS) interface through which the partitions and the vector operand of SpMV are transferred from the memory to FPGA, and (iii) the high-level three-stage pipeline (Figure 6.2, ①) implemented in the FPGA that receives the partitions in the memory-read stage, processes them in the compute stage (i.e., SpMV block in Figure 6.2), and streams the partial output vector back to the memory in the memory-write stage. The input buffer contains a partition compressed in a particular format (e.g., it contains values, offsets, and column indices for CSR). The compute stage itself comprises a two-stage pipeline, including decompress and dot-product stages. The block of SpMV iteratively creates dense non-zero rows in the first stage (Figure 6.2, ②). Then, the second stage (Figure 6.2, ③) performs a dot-product between the result of the first stage and the vector operand of SpMV. We implement the dot-product as

an array of multipliers connected to a balanced adder tree. Since the output of the SpMV is a vector (not necessarily sparse), we do not include a recompression stage in hardware.

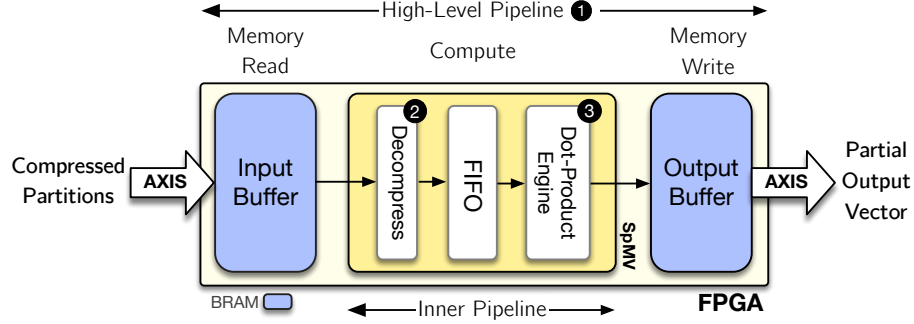


Figure 6.2: *The architecture of our evaluation platform:* Streaming the compressed partitions of a sparse matrix from the memory to FPGA through AXI stream interfaces and processing them (i.e., SpMV) in a pipeline. The *decompression* component varies based on sparse format.

To take advantage of stream accesses to memory and a parallel compute engine, the key component of Ascella is a lightweight microarchitecture for creating *dense* rows (Figure 6.2, ② details of which are shown in Figure 6.3). This microarchitecture implements deterministic parallel accesses to the values and indices and significantly reduces the decompression latency by just applying a lightweight logical operation (i.e., AND) to generate addresses. The mechanism of Figure 6.3 is shown in Figure 6.4 by illustrating the steps to decompress all non-zero rows of the example in Figure 6.1. At each step, we use *read indices* to read the *column indices* ①. The minimum of column indices defines *column index* ② of the next non-zero row. *Column index* is used to create a binary mask. The *values* corresponding to ones in the mask are selected to participate in creating a *dense row* ③, which is the input of dot-product. The mask is also used for updating the *read indices* ④.

6.2 Evaluation

6.2.1 Experimental setup

We implement the microarchitecture of Ascella and the baselines using Xilinx® Vivado® HLS. We use relevant *#pragma* as hints to describe our desired microarchitectures in C++.

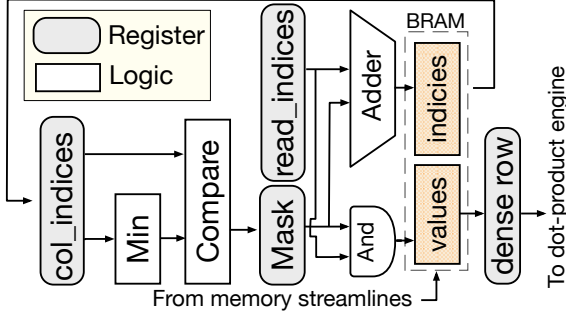


Figure 6.3: The microarchitecture of Ascella for decompression.

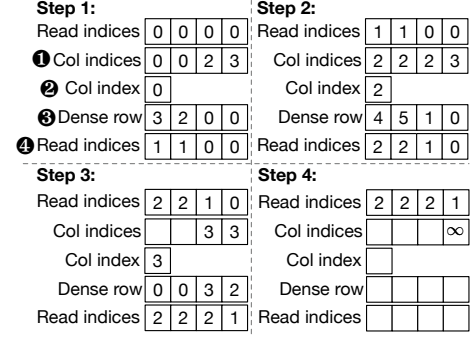


Figure 6.4: The steps of decompression using Ascella.

We synthesize the designs on ZYNQ XC7Z020 FPGA and report post-synthesis performance numbers and resource utilization. In the following, we explain our tailored HLS implementation of the decompression mechanisms for the seven target sparse formats.

CSR: As Listing 6.1 shows, since CSR uses three vectors (i.e., offsets, column indices, and values) to represent a sparse matrix, for decompressing a non-zero row, we need to first access the offsets (Listing 6.1, line 7), and then, based on `numVal`, we can read as many column indices and values as required (Listing 6.1, line 10). As a result, decompression from the CSR format is likely to be compute-bound because of the overhead of one extra access to BRAM. Additionally, to retrieve the column indices and values, *since accesses to the BRAM blocks are sequential*, we do not know in advance which elements of column indices and values are going to be accessed. Thus, we cannot partition and allocate those two vectors across the blocks of BRAM to guarantee parallel accesses. Because of the sequential accesses in a non-zero row, the latency of decompressing a row depends on the number of non-zero elements in that row. By assuming that we stream the offsets and column indices using two streamlines in parallel, the one with more non-zero elements (longer one) defines the latency of memory access. To reduce the negative impact of the accesses to offsets on performance, we pipeline this progress to concurrently create non-zero rows (if more than one).

```
1 function decompressCSR(A, readInx, oldInx)
2     // A in CSR: offsets[OFFSET_LENGTH]
```

```

3      //          colInx[COL_INX_LENGTH]
4      //          values[VAL_LENGTH]
5      // readInx: current row index
6      // oldInx: last row index
7      numVal = offsets[readInx] - offsets[readInx-1]
8      for i=0 to numVal:
9          #pragma HLS pipeline
10         drow[colInx[oldInx+i]] = values[oldInx+i]
11 return drow

```

Listing 6.1: CSR-decompression HLS pseudo code

BCSR: As Listing 6.2 shows, the decompression of BCSR is similar to that of CSR, whereas instead of individual non-zero elements, the non-zero blocks are processed. To initiate the accesses to column indices and values, one extra access to the offsets is required per each row of blocks (Listing 6.2, line 9). The advantage of BCSR over CSR is that we can distribute the values and column indices over BRAM blocks and access their elements in parallel, for which, as lines 1 and 2 in Listing 6.2 show, we completely partition the values and colInx across their second dimension before calling the decompression function. This allows us to unroll the for loop (line 12), the iterations of which access different BRAM blocks in parallel. The downsides of BCSR, however, are (i) the overhead of transferring zero elements in the non-zero blocks and (ii) processing all the rows in the non-zero blocks whether they (the rows) are all zero or not. The latency of decompressing the blocks in a row depends on the number of non-zero *blocks* in that row. On the other hand, since the values has the longest length (compared to offsets and colInx), transferring values defines the memory latency.

```

1 #pragma HLS array_partition variable=values dim=2
2 #pragma HLS array_partition variable=colInx dim=2
3 function decompressBCSR(A, readInx, oldInx)
4     // A in BCSR: offsets[OFFSET_LENGTH]
5     //          colInx[COL_INX_LENGTH]

```

```

6 //          values[VAL_LENGTH][VAL_WIDTH]
7 // readInx: current row index
8 // oldInx: last row index
9 numBlocks = offsets[readInx] - offsets[readInx-1]
10 for i=0 to numBlocks:
11     for j=0 to VAL_WIDTH:
12         #pragma HLS unroll
13         drows[j / BLOCK_LENGTH][colInx[oldInx + i]
14             + j mod BLOCK_LENGTH] = values[oldInx + i][j]
15 return drows

```

Listing 6.2: BCSR-decompression HLS pseudo code

CSC: Listing 6.3 shows the pseudo-code of CSC decompression, in which the *columns* are compressed. On the contrary, the hardware requires *rows* of the matrix for performing SpMV. Because of this mismatch, the decompression mechanism must iteratively traverse all the columns of the matrix to find the values corresponding to the current row (Listing 6.3, line 12). Although this mismatch makes the decompression inefficient, we still include this extreme case in our evaluation to explore how much performance is hurt if the format and the hardware are not aligned.

```

1 function decompressCSC(A, readInx)
2     // A in CSC: offsets[OFFSET_LENGTH]
3     //          rowInx[ROW_INX_LENGTH]
4     //          values[VAL_LENGTH]
5     // readInx: current row index
6     numVal = offsets[colInx] - offsets[Inx-1]
7     for i=0 to CSC_ROW_INX_LENGTH
8         && colInx < CSC_OFFSETS_LENGTH:
9             startInx = i
10            #pragma HLS pipeline
11            while i < startInx + numVal:
12                if rowInx[i] == read_inx:

```

```

13         drow[colInx-1] = values[i]
14         break
15         i++
16         i = offsets[colInx++]
17 return drow

```

Listing 6.3: CSC-decompression HLS pseudo code

LIL: The LIL decomposition (Listing 6.4) avoids extra accesses to BRAM and enables *deterministic* parallel accesses to the BRAM blocks for decompressing the non-zero rows of a sparse matrix. Since the columns of `values` and `indices` can always be accessed in parallel, we partition both of them (Listing 6.4, lines 1 and 2) before calling decomposition. As a result, no extra read access is required to determine the number of next read accesses. Thus, the latency of processing a matrix depends on the number of non-zero rows. In other words, creating a non-zero row consists of the latency of one BRAM access (since the accesses are parallel) plus the latency for creating the input of the dot product, which is done by a simpler logic compared to those of CSR or BCSR. To recognize the end of the non-zero rows, one additional BRAM access is required. Memory latency for LIL is defined by the number of non-zero rows, the size of rows, and transferring one additional row for indicating the end of non-zero rows.

```

1 #pragma HLS array_partition variable=values dim=2
2 #pragma HLS array_partition variable=Inx dim=2
3 function decompressLIL(A, readInx[], oldInx)
4     // A in LIL: values[HEIGHT][WIDTH]
5     //           Inx[HEIGHT][WIDTH]
6     // readInx: current row index
7     // oldInx: last row index
8     minInx = inf
9     for i=0 to WIDTH:
10         #pragma HLS pipeline
11         if readInx[i]<HEIGHT && Inx[readInx[i]][i]<minInx:

```

```

12     minInx = Inx[readInx[i]][i]
13     for i=0 to WIDTH:
14         #pragma HLS unroll
15         if Inx[readInx[i]][i] == minInx:
16             drow[i] = values[readInx[i]][i]
17             readInx[i]++
18 return drow

```

Listing 6.4: LIL-decompression HLS pseudo code

ELL: Similar to LIL, the representation of matrix A in ELL (Listing 6.5) includes values and indices that can be accessed in parallel and thus are partitioned and distributed over BRAM blocks (Listing 6.5, lines 1 and 2), which subsequently allows unrolling the for loop for parallel processing (line 7). The difference between LIL and ELL, however, is the direction of compression. Although the direction of compression in ELL enables a simple assignment shown in line 8, it prevents skipping the all-zero rows that in turn can cause a performance drop. Since we completely unroll the for loop (line 7), reducing ELL_MAX_COMP_ROW_LENGTH in the ELL implementation and using optimizations such as ELL-COO only impact the resource utilization of FPGA, not the performance.

```

1 #pragma HLS array_partition variable=values dim=2
2 #pragma HLS array_partition variable=indices dim=2
3 function decompressELL(A)
4     // A in ELL: values[ELL_MAX_COMP_ROW_LENGTH]
5     //             indices[ELL_MAX_COMP_ROW_LENGTH]
6     for i=0 to ELL_MAX_COMP_ROW_LENGTH:
7         #pragma HLS unroll
8         drow[indices[i]] = values[i]
9 return drow

```

Listing 6.5: ELL-decompression HLS pseudo code

COO: As Listing 6.6 shows, since COO saves tuples for representing a matrix, its de-compression mechanism is pretty straightforward, including a simple assignment, shown

in Listing 6.6, line 7. The downside of COO, however, is that we do not know in advance how many elements exist in each row. Thus, we cannot partition and allocate the vector of `tuples` across the blocks of BRAM to guarantee parallel accesses. For the same reason, we pipeline the for loop (line 5) rather than unrolling it. The same procedure is also applicable to DOK.

```

1 function decompressCOO(A, row)
2   // A in COO: tuples[COO_NUM_TUPLES][3]
3   // row: the current row
4   for i=0 to COO_NUM_TUPLES:
5     #pragma HLS pipeline
6     if (tuples[i][0] != inf && tuples[i][0] == row):
7       drow[tuples[i][1]] = tuples[i][2];
8 return drow

```

Listing 6.6: COO-decompression HLS pseudo code

DIA: Line 7 of Listing 6.7 shows the pseudo-code for the decompression mechanism of DIA, the most domain-specific format. DIA saves matrix *A* as *diags*, a two-dimensional matrix including all non-zero diagonals of matrix *A*. The first element of each row of *diags* indicates the diagonal number. To decompress the rows of matrix *A*, the decompression function traverses all rows of *diags* to find the elements corresponding to the current *row*. To this end, we use two helper functions, *DiaInxForRow* (line 1) and *IsRowOnDiagonal* (line 4). As the decompression mechanism suggests, although in terms of memory footprint DIA should be beneficial for diagonal matrices, its decompression mechanism is not quite compatible with even a simple computation such as a fine-grained parallel SpMV, which subsequently requires rows of the matrix. Such an overhead worsens when non-zero elements are scattered over multiple diagonals but do not completely fill them.

```

1 function DiaInxForRow(row, d)
2   return (row + d < row) ? row + d : row

```

```

3
4 function IsRowOnDiagonal(row, d)
5     return d <= WIDTH - 1 - row && d >= -row
6
7 function decompressDIA(A, row)
8     // A in DIA:
9     //   diags[NUM_DIAGONALS][MAX_DIAGONAL_LEN]
10    // row: the current row
11    for i=0 to NUM_DIAGONALS:
12        #pragma HLS pipeline
13        d = diags[i][0]
14        if (!IsRowOnDiagonal(row, d)) continue
15        // column index is the row + d
16        // + 1 since 0 element is the diagonal number
17        drow[row+d] = diags[i][DiaInxForRow(row,d)+1]
18 return drow

```

Listing 6.7: DIA-decompression HLS pseudo code

Configurations and Workloads: The baselines and Ascella use similar memory stream interfaces to communicate with external DDR3 memory, and utilize the same dot-product engine, and, only their decompression logic differ. Inputs and output of the accelerators are transferred through the AXI stream interface. The clock frequency is set to 100 MHz. All computations are on 32-bit integers. For BCSR, the sub-block size is four. We run SpMV on various-size matrices, with applications in scientific and graph problems.

Besides real-world matrices, our workloads consist of two groups of synthetic sparse matrices. The first group includes *randomly generated* sparse matrices, the density of which varies from 0.0001 to 0.5. We generate the denser random matrices (i.e., the density of 0.1 to 0.5) as a representation for those in machine learning applications. On the other hand, the more sparse random matrices (i.e., density between 0.0001 to 0.01) represent scientific and graph applications with no particular structure. The second group of

our synthetic sparse matrices denotes the common structure in sparse matrices: *diagonal and band matrices*. A band matrix is a sparse matrix, the non-zero entries of which are confined to a diagonal band, including the main diagonal and more than one diagonal on each side. The width of a band matrix is the number k such that $a_{i,j} = 0$ if $|i - j| > k/2$. We generate and evaluate band matrices of size 8000 with widths of 2, 4, 16, 32, and 64. Numerical problems in higher dimensions often lead to band matrices (e.g., a PDE on a square domain). A type of band matrices consisting of only the main diagonal (i.e., $k = 1$) is called a diagonal matrix. Diagonal matrices also occur in many fields of linear algebra.

Metrics & Hyperparameters: We evaluate the performance implications of sparse formats using the metrics introduced in the following. First, we define σ as a metric to measure the latency **overhead of decompression**:

$$\sigma = \frac{T_{decomp} + nnz_rows \times T_{dot}}{p \times T_{dot}} \quad (6.1)$$

in which T_{decomp} indicates decompression latency, which consists of latency for BRAM accesses and logic, nnz_rows are non-zero rows for which we must perform a dot-product, each taking T_{dot} , and p is the partition size. As a result, for the dense format, $\sigma = 1$. Besides σ , we measure the breakdowns of latency: (i) **memory latency**, the time to transfer a compressed partition (data and metadata) to FPGA and buffer it in the BRAM; and (ii) **computation latency** consisting of decompression, dot-product, and necessary BRAM accesses. Furthermore, seeking an appropriate sparse format for achieving balanced streaming, we proposed evaluating a **balance ratio**, which we define as the average ratio of memory latency to compute latency for all non-zero partitions. The balanced ratio of perfectly balanced streaming would be one. An imbalance streaming leads to idle computation or pauses in data transfer.

We also evaluate **throughput**, defined as bytes processed per second, which reflects the bubbles in the streaming pipeline caused by imbalance streaming (balance ratio $\neq 1$). Besides throughput, we compare the **memory-bandwidth utilization**, the ratio of useful data over all transmitted data (i.e., useful data plus metadata). Our other metrics for

the full design-space exploration are **resource utilization** and dynamic/static **power consumption**. By resource utilization, we mean the percentage of FPGA resources used by all components (entire Figure Figure 4.6). We evaluate the aforementioned metrics while varying the hyperparameters including practical partition sizes of 8, 16, and 32, and the width of 2, 4, 8, 16, 32, and 64 for band matrices to represent realistic matrices, and the density of random matrices from 0.0001 to 0.5.

6.2.2 The Overhead of Decompression

This section explores σ , the latency overhead of decompression for SuiteSparse, random, and structured band matrices in Figure 6.5, Figure 6.6, and Figure 6.7, respectively. In Figure 6.5, the bars lower than one illustrate faster computation than the dense format. The overhead of the dense baseline, for which $\sigma = 1$, is computing and transferring zero entries, and the overhead of all sparse formats (only for computations) goes to the decompression. As Figure 6.5 shows, the overhead of sparse formats can, in some cases, exceed that of the dense format. The worst-case scenario of decompression occurs with the CSC format because the orientation of data is opposite to that of the mechanism in hardware.

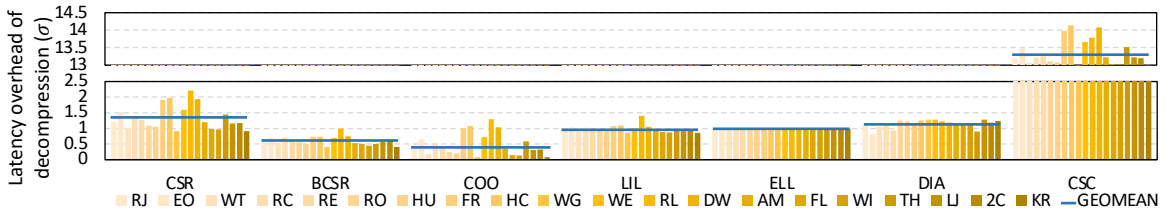


Figure 6.5: Decompression overhead for SuiteSparse: comparing the latency overhead, σ (lower is better), of seven sparse formats for partition size of 16×16 . A darker color indicates less sparsity (i.e., higher density).

From Figure 6.5, we do not observe any relationship between the density (darkness of the bars in Figure 6.5) and σ in highly sparse matrices. Thus, Figure 6.6 clarifies such a relationship for a wider range of density based on our randomly generated synthetic workloads. Likewise, Figure 6.7 shows the latency of band matrices when the width increases.

As the two figures illustrate, although the σ of all formats increase with density and width of band matrices, it more dramatically increases for COO, CSR, and CSC. Besides, the time to reconstruct the rows of a matrix from a column-oriented compression format (i.e., CSC) leads to up to $21\times$ and $30\times$ slower computation than if we were to process all zero entries of the dense format, respectively, for random and band matrices. In such cases, preprocessing the sparse data to a format compatible with a hardware accelerator is highly suggested.

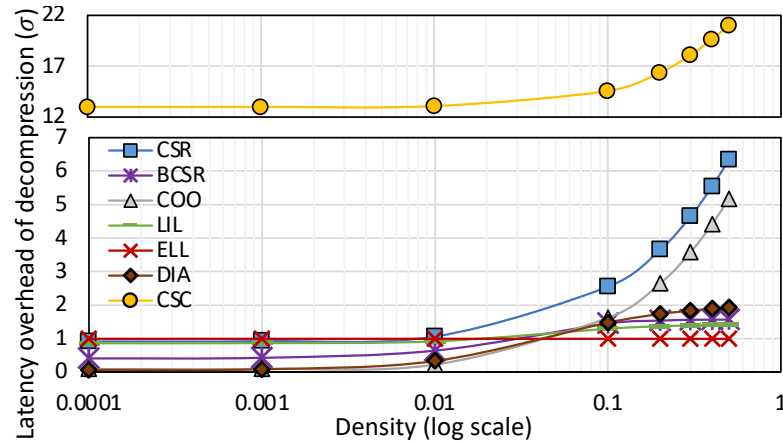


Figure 6.6: Decompression overhead for random matrices: comparing the σ (lower is better) of seven formats for 16×16 partitions when density varies from 0.0001 to 0.5.

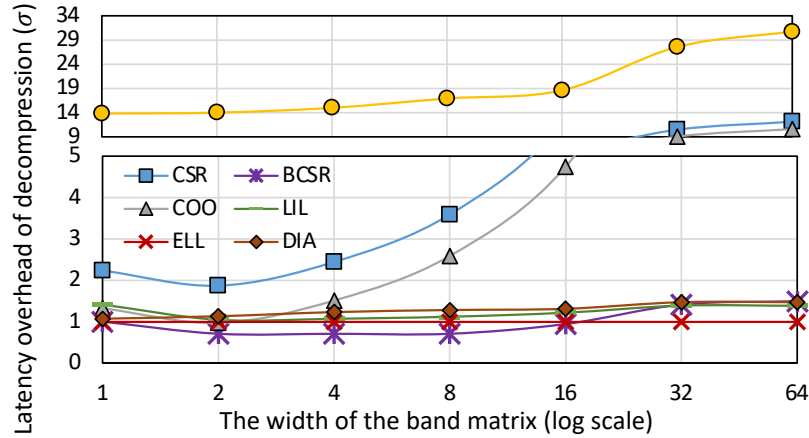


Figure 6.7: Decompression overhead for band matrices: comparing the latency overhead, σ , of seven sparse formats for partition size of 16×16 when the width varies from 1 to 64.

Figure 6.8 illustrates the impact of partition size on σ . In all workloads (i.e., SuiteSparse, random, and structured), the computation latency of ELL is proportional to that of

the dense format and does not change with the pattern of sparsity. This is because, in ELL, we are still processing a whole non-zero matrix regardless of its individual entries. However, since the length of these new squares (in our case, six) is smaller than that of the original dense partitions (8, 16, or 32), the computation latency of ELL decreases as the partition size increases. Seeking a relatively generic sparse format that can provide moderate computation latency for random and structured matrices, BCSR could be a fair option. However, it is not as good for random matrices when the partition size increases. This is because of the additional dot products that must be done per each non-zero block regardless of the individual values of the entries.

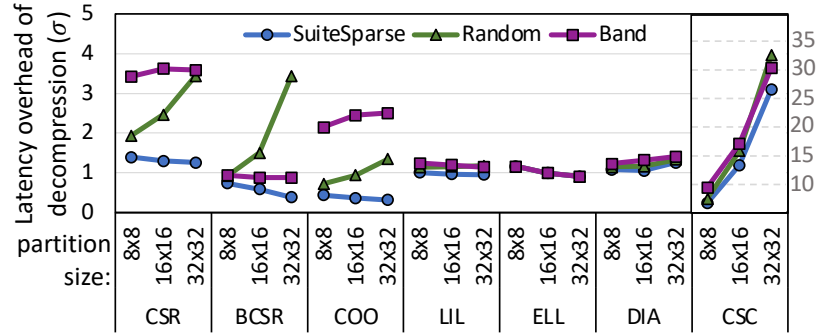


Figure 6.8: Decompression overhead for various partition sizes: comparing the average σ (lower is better) of seven sparse formats for three types of workloads (SuiteSparse, random, band) and partition sizes of 8, 16, and 32.

6.2.3 Latency and Balance Ratio

Since memory accesses and computation are pipelined, the sum of their *maximum for each partition* defines the total latency. Thus, the latency overhead (discussed in subsection 6.2.2), which stems from only the computation, does not provide any information about which one (i.e., computation or memory) defines the total latency. Details in that regard are discussed in this section. Figure 6.9 shows both memory and compute latency and thus implicitly shows the balance ratio: points below the *balance* line have a balance ratio smaller than one.

Since only non-zero entries of non-zero partitions are transmitted, the latency to trans-

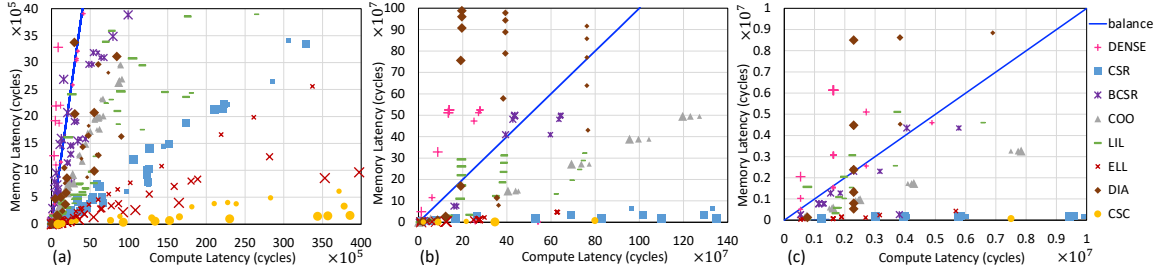


Figure 6.9: Balance ratio: the relationship between the memory and compute latency for various partition sizes indicated by the size of markers for (a) SuiteSparse, (b) random workloads, and (c) band matrices. The blue line indicates balance ratio = 1.

mit data and metadata (i.e., memory latency) for all sparse formats is much lower than that for the dense format, as expected. The computation latency of sparse formats, however, is not always lower than for the dense format. While some of the frequently used formats such as CSR, CSC, and DIA have been successful in lowering the memory latency, their computation latency is higher than the baseline, which negates their benefits. Other formats such as LIL and ELL also do well in reducing memory latency (i.e., data-transfer time) but have similar computations as the baseline. For instance, the computation latency of LIL is defined by the longest column; hence, in some cases, it is the same as or more than that of dense format. For ELL, on the other hand, when the width of the ELL matrix is slightly smaller than the width of the original partition (e.g., the 8×8 case), the computation latency of ELL is just slightly higher than dense format because of the overhead of decompression, even though it is small. Similar to random and structured matrices, the CSC format is the slowest for SuiteSparse workloads with up to $27\times$ higher latency compared to the dense baseline. All in all, Figure 6.9 suggests that in terms of latency, COO or BCSR could be appropriate candidates to be used for diverse matrices from scientific and graph applications, even though in some cases they perform as good as the dense format.

As Figure 6.9 shows, for all types of matrices (i.e., SuiteSparse, random, and band), the balance ratio of dense format is higher than most of the sparse formats. This is because the zero entries impact both memory and computation latency. In fact, the balance ratio of dense format is closer to one (i.e., the perfect case) – but it moves toward a memory-

bound as partition size, indicated by marker size, increases. In some formats, such as LIL, increasing the partition size helps to achieve a better balance, while in some others, such as ELL, it is the opposite. In random and structured band sparse matrices (Figure 6.9b and Figure 6.9c), higher density and/or larger bandwidth of band matrices leads to a memory bottleneck for BCSR, LIL, and DIA. In such cases, if adding more memory bandwidth to the system is possible, using BCSR or LIL for less sparse application (e.g., for the inference of neural networks) or using DIA for applications with diagonal/band matrices is suggested. Otherwise, COO seems to offer a reasonable balance for various densities as well as the varieties of band matrices. The same hypotheses for balance ratio are also applicable for the more diverse SuiteSparse workloads, as shown in Figure 6.9a.

6.2.4 Throughput and Bandwidth Utilization

This section studies throughput and memory bandwidth utilization. First, Figure 6.10 explores the relationship between throughput and the total time to process an 8000×8000 matrix. The following parameters contribute to throughput: (i) the total processed data consisting of data and metadata and (ii) the total time to process, which is the maximum of memory latency (data-transfer time) and computation latency for each partition. As a result, in a sparse format such as ELL in which both total latency and data grow with the same pace, throughput does not change with latency (this is also the case for the dense baseline). For all formats but ELL, throughput increases with latency and then reaches a maximum. As Figure 6.10 suggests, BCSR, LIL, and DIA reach a higher throughput compared to the other four formats. Besides, for all formats but CSC, increasing partition size, shown by the thickness of lines in Figure 6.10, results in higher throughput because both latency and data decrease as the partition size increases. Since throughput does not reflect the impact of transmitting and computing *useful* data, we study throughput along with the utilization of memory bandwidth.

Memory bandwidth utilization and its relationship with density, width of band matri-

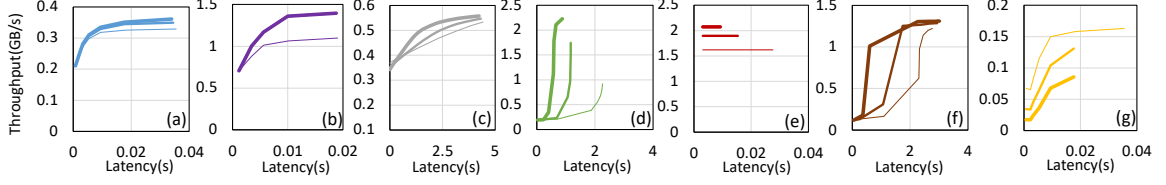


Figure 6.10: Throughput vs. latency: comparing the throughput and the average time to apply SpMV on an 8000×8000 : (a) CSR, (b) BCSR, (c) COO, (d) LIL, (e) ELL, (f) DIA, and (g) CSC. Thicker lines indicate larger partition sizes.

ces, and partition size are shown in Figure 6.11, Figure 6.12, and Figure 6.13, respectively. As they all indicate, the memory bandwidth utilization of COO is always 0.3 since it always transmits two indices per one non-zero entry. Besides, as Figure 6.12 indicates, the memory bandwidth utilization of DIA for diagonal matrices is close to one – the slight difference occurs because of saving the diagonal number for the main diagonal. As partition size grows, this memory bandwidth utilization approaches full utilization. However, for other band matrices, we see that the DIA format does not offer better memory bandwidth compared to *more generic* formats such as COO, ELL, or LIL, among which LIL is a better candidate to cover more extreme sparseness as well as a wider variety of random matrices (Figure 6.11) while offering a better balance ratio at larger partitions compared to COO and ELL. Finally, Figure 6.13 demonstrates that, as expected, for all formats but COO, the memory bandwidth utilization of denser matrices (density > 0.1) and structured ones is higher than that of extremely sparse matrices (e.g., SuiteSparse).

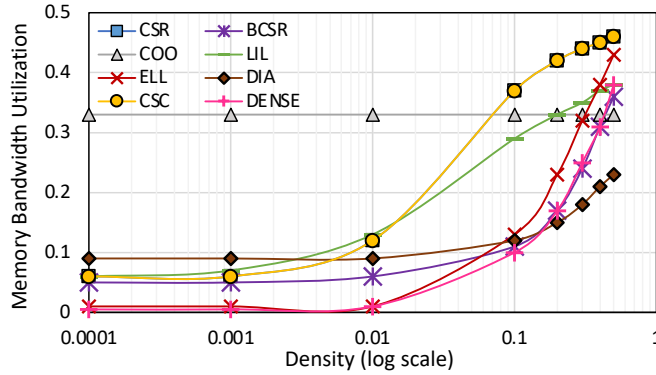


Figure 6.11: Memory bandwidth utilization for random matrices: comparing seven sparse formats for partition size of 16×16 when density varies from 0.0001 to 0.5.

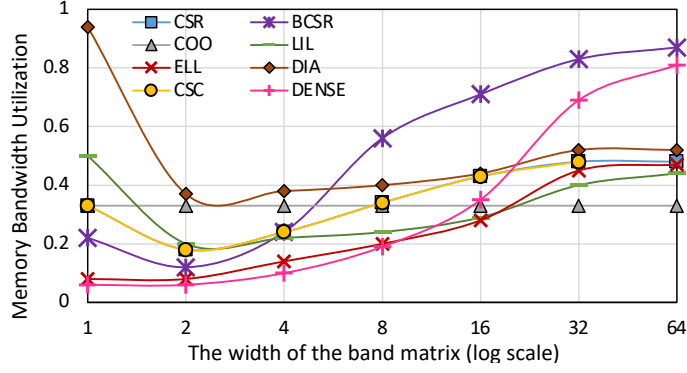


Figure 6.12: Memory bandwidth utilization for band matrices: comparing seven sparse formats for partition size of 16×16 when the width varies from 1 to 64.

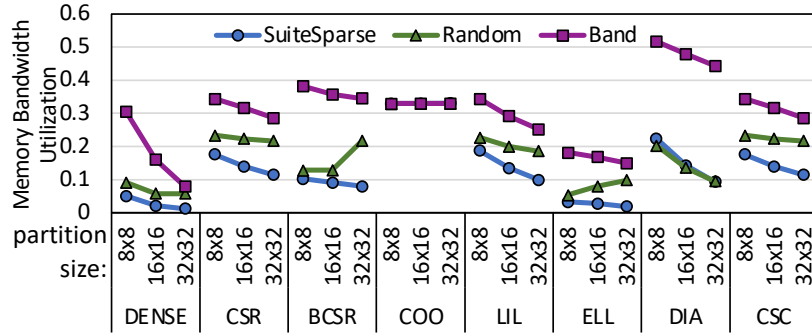


Figure 6.13: *Memory bandwidth utilization for various partition sizes*: comparing the average memory bandwidth utilization (higher is better) on SuiteSparse, random, band workloads for partition sizes of 8, 16, and 32.

6.2.5 Resource Utilization and Power Consumption

Table 6.1 compares FPGA resource utilization and dynamic power consumption. We must dedicate enough BRAM blocks to envision the worst-case scenarios even though they occur rarely. The other factor impacting the BRAM utilization is the degree of parallelism. To enable parallelism, we partition the matrices and distribute them to BRAM blocks. Because of these two factors, we see that CSR and CSC utilized the lowest number of BRAM blocks, whereas BCSR utilizes the same blocks as the dense implementation does. In some cases, such as ELL, smaller partitions (i.e., 8 and 16) use more flip flops (FFs) compared to larger partitions (i.e., 32). This is because, in a small partition size, the buffering is automatically implemented using FFs rather than BRAM blocks. It is also demonstrated by the fewer BRAM blocks utilized by the 8×8 ELL.

Table 6.1: Resource utilization and the total dynamic power consumption for three partition sizes (8, 16, and 32).

part. size:	BRAM_18K			FF ($\times 1000$)			LUT ($\times 1000$)			DY Power(W)		
	8	16	32	8	16	32	8	16	32	8	16	32
DENSE	8	16	32	1.5	1.9	4.3	0.7	0.7	1.2	0.02	0.08	0.03
CSR	2	2	8	0.7	0.8	3.8	0.9	0.9	1.1	0.04	0.04	0.07
BCSR	8	16	32	1.6	2.4	4.4	1.2	1.4	2.2	0.05	0.06	0.06
CSC	1	1	9	0.9	1	2.7	1	1.2	1.1	0.01	0.05	0.03
LIL	4	4	6	2.9	5.8	9.1	1.6	2.7	4.8	0.05	0.08	0.07
ELL	1	7	9	2	3.2	0.9	0.9	1	0.8	0.06	0.10	0.06
COO	3	3	8	1.8	1.3	3.2	1.2	2.5	5.4	0.02	0.04	0.04
DIA	3	3	11	2.2	5	9.2	1.5	2.8	4.6	0.07	0.12	0.05
Total	140			106.4			53.2			N/A		

The dynamic power consumption listed in Table 6.1 suggests that while larger partition sizes cause higher power consumption in some formats (i.e., CSR, BCSR, COO, and LIL), for the others (i.e., dense, CSC, ELL, and DIA), the maximum power is consumed at the 16×16 partition size and the minimum case may occur at 8×8 (e.g., dense and CSC) or at 32×32 (e.g., ELL and DIA). To clarify, Figure 6.14a, Figure 6.14b, and Figure 6.14c illustrate the dynamic power consumed by logic, BRAM, and signals, respectively. As Figure 6.14 shows, the power consumption of logic always increases or stays steady as partition size increases, while that of BRAM may decrease (e.g., dense and BCSR). Therefore, comparing Figure 6.14 against the dynamic power listed in Table 6.1 indicates that the trend of overall dynamic power consumption partially depends on BRAM, but more generally follows the same trend as the power consumption of signals (Figure 6.14c). By evaluating total latency and power consumption together, we see that for SuitSparse matrices, not only does COO consume the least dynamic power, but also it is the fastest in terms of total latency. However, if achieving high throughput at lower power is the goal, BCSR is a better fit. On the other hand, for structured matrices, LIL and ELL are the fastest in terms of latency and throughput, among which ELL performs better for band matrices with wider bandwidths and consumes less power. The static power consumption of dense, CSR, BCSR, LIL, and ELL is 0.121W and that of CSC, COO, and DIA is 0.103W. The static energy, which depends on time, can be an issue for those slower sparse formats that require

less amount of dynamic energy.

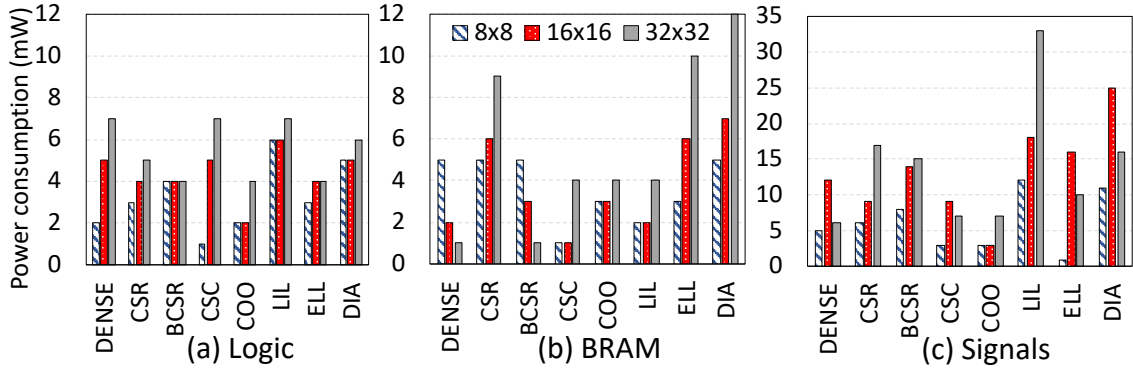


Figure 6.14: Dynamic power consumption: (a) logic, (b) BRAM, and (c) signals.

6.3 Summary

This chapter showed that the primary challenge with sparse matrices has been efficiently storing and transferring data, for which many sparse formats have been proposed to significantly eliminate zero entries. Such formats, essentially designed to optimize memory footprint, may not be as successful in performing faster processing. In other words, although they allow faster data transfer and improve memory bandwidth utilization the classic challenge of sparse problems – their decompression mechanism can potentially create a *computation* bottleneck. Not only is this challenge not resolved, but also it becomes more serious with the advent of domain-specific architectures, as they intend to more aggressively improve performance. The performance implications of using various formats along with DSAs, however, has not been extensively studied by prior work. To fill this gap of knowledge, this chapter characterized the impact of using seven frequently used sparse formats on performance, based on a DSA for sparse matrix-vector multiplication, implemented on an FPGA using HLS tools, a growing and popular method for developing DSAs. Seeking a fair comparison, we tailor and optimize the HLS implementation of *decompression* for each format. We thoroughly explored diverse metrics, including decompression overhead, latency, balance ratio, throughput, memory bandwidth utilization, resource utilization, and power consumption, on a variety of real-world and synthetic sparse workloads.

CHAPTER 7

CONCLUSIONS

This thesis proposed solutions to deal with the challenges in efficiently executing sparse problems. First, we proposed Fafnir, a DDR-based NDP solution for accelerating embedding lookup, the bottleneck-prone task in recommendation systems. The key component of Fafnir is a near-memory intelligent reduction tree, which provides a generic solution for any *sparse gathering*. Besides embedding lookup and SpMV, sparse gathering is also a required function in numeric algebra such as matrix inversion and differential-equation solvers. The particular patterns of computation in such applications necessitate some additional connections in the structure of a tree, which will be envisioned in our future work. The same idea of Fafnir can also be integrated with High Bandwidth Memory (HBM) by connecting the leaf PEs to the 32 pseudo channels rather than the ranks.

As another preliminary solution, this thesis proposed Alrescha, a generic accelerator for scientific and graph problems. We showed that Alrescha not only can accelerate graph algorithms efficiently by using a data-driven execution model, which eliminates transferring metadata during the run time but also is reconfigurable to handle dependency patterns in most of the scientific algorithms. Alrescha mathematically transforms the algebraic operations in iterative PDE solvers to increase the chance of parallelism in scientific problems. We have also studied compression formats and learned that to be effective in *streaming* sparse data, the order of non-zero values and the decompression mechanism are important. We proposed the storage format of Alrescha, which helps to accelerate sparse problems by enabling stream accesses to memory.

This thesis also introduced Lodestar, a novel approach for pruning DNNs based on the requirements of *systolic arrays*. We focused on the importance of the *distribution* of non-zero values in sparse DNN models rather than their quantity when we use systolic arrays

for DNN inference. Unlike prior work, Lodestar examines the correlation among the filters rather than the individual filters to increase the chances of modifying the distribution of non-zeros. To handle the locally-dense data, created by Lodestar, we introduced Eridanus, a streaming accelerator with the core of a stationary systolic array. Eridanus employs a very simple indexing logic.

Last but not least, this thesis would lead architects to knowingly choose the required sparse format and tailor their FPGA designs for sparse applications. Figure 7.1 summarizes all the six studied metrics for three group of sparse workloads by normalizing each metric to its maximum achieved number so that "1" represents the best case and "0" represents the worst case. Here, we overview some insights:

- Unlike a common belief, the memory bandwidth is not always the bottleneck; hence the performance sparse problems cannot always be improved by simply adding more memory bandwidth to the system. Thus, when using a format such as CSR to efficiently use storage, a lower-bandwidth low-cost memory is sufficient. Otherwise, the implementation of the computations must be further improved (if possible).
- Although in scientific computing and graph analytics (Figure 7.1a), the common patterns of sparse matrices are diagonal and band, our study shows that a non-specialized format such

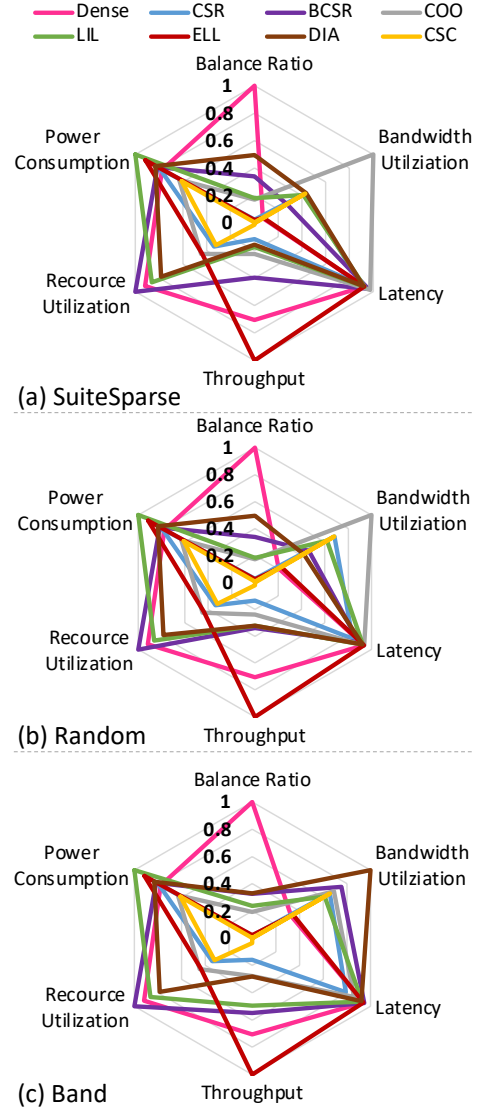


Figure 7.1: Comparing sparse formats for: (a) SuiteSparse, (b) random, and (c) band matrices. 1 and 0 show best and worst, respectively.

as COO performs faster and better utilizes the memory bandwidth compared to a specialized format such as DIA. This is because of the compatibility of more generic formats with a generic hardware for common computations. Besides, a generic format better tolerates the variations in the distribution of non-zero entries. If power consumption and FPGA resource utilization must also be considered, LIL or BCSR are other candidates.

- For structured band matrices (Figure 7.1c), a pattern-specific format such as DIA, near-perfectly utilizes the memory bandwidth and does it better as the partition size increases. However, to allow such utilization to effectively impact the other performance metrics, the computation engine must also be tailored to the format if DIA must be used in a particular application. Otherwise, the mismatch would create a computation bottleneck.
- For less sparse (density > 0.1) applications such as the inference of neural network, optimizations beyond simple partitioning of size 8×8 or at most 16×16 hurt the performance even though it might help reduce the memory footprint (possibly, not too much). Extracting the non-zero partitions from the neural network can be done with the aid of structure pruning schemes [89, 92, 93, 51, 19].

REFERENCES

- [1] J. C. Beard, “The sparse data reduction engine: Chopping sparse data one byte at a time”, in *Proceedings of the international symposium on memory systems*, 2017, pp. 34–48.
- [2] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. Costa, J. Doi, C. Evangelinos, *et al.*, “Active memory cube: A processing-in-memory architecture for exascale systems”, *Ibm journal of research and development*, vol. 59, no. 2/3, pp. 17–1, 2015.
- [3] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, “Graphq: Scalable pim-based graph processing”, in *Proceedings of the 52nd annual ieee/acm international symposium on microarchitecture*, 2019, pp. 712–725.
- [4] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, “Graphp: Reducing communication for pim-based graph processing with efficient data partition”, in *HPCA*, IEEE, 2018, pp. 544–557.
- [5] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, “Lodestar: Creating locally-dense cnns for efficient inference on systolic arrays”, in *Proceedings of the 56th annual design automation conference 2019*, ACM, 2019, p. 233.
- [6] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, “Graphpim: Enabling instruction-level pim offloading in graph computing frameworks”, in *High performance computer architecture (hPCA), 2017 IEEE international symposium on*, IEEE, 2017, pp. 457–468.
- [7] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing”, in *Proceedings of the 42nd annual international symposium on computer architecture*, 2015, pp. 105–117.
- [8] B. Asgari, R. Hadidi, N. S. Ghaleshahi, and H. Kim, “Pisces: Power-aware implementation of slam by customizing efficient sparse algebra”, in *2020 57th ACM/IEEE design automation conference (dac)*, IEEE, 2020, pp. 1–6.
- [9] J. G. Lewis and H. D. Simon, “The impact of hardware gather/scatter on sparse gaussian elimination”, *Siam journal on scientific and statistical computing*, vol. 9, no. 2, pp. 304–311, 1988.
- [10] P. Covington, J. Adams, and E. Sargin, “Deep neural networks for youtube recommendations”, in *Proceedings of the 10th ACM conference on recommender systems*, 2016, pp. 191–198.

- [11] C. A. Gomez-Urbe and N. Hunt, “The netflix recommender system: Algorithms, business value, and innovation”, *Acm transactions on management information systems (tmis)*, vol. 6, no. 4, pp. 1–19, 2015.
- [12] U. Gupta, X. Wang, M. Naumov, C.-J. Wu, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, B. Jia, H.-H. S. Lee, *et al.*, “The architectural implications of facebook’s dnn-based personalized recommendation”, 2020.
- [13] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, *et al.*, “Deep learning recommendation model for personalization and recommendation systems”, *Arxiv preprint arxiv:1906.00091*, 2019.
- [14] H.-J. M. Shi, D. Mudigere, M. Naumov, and J. Yang, “Compositional embeddings using complementary partitions for memory-efficient recommendation systems”, *Arxiv preprint arxiv:1909.02107*, 2019.
- [15] *COVID-19 High Performance Computing (HPC) Consortium*, <https://covid19-hpc-consortium.org/projects>, [Online; accessed October-2020], 2020.
- [16] J. Dongarra, M. A. Heroux, and P. Luszczek, “Hpcg benchmark: A new metric for ranking high performance computing systems”, *Knoxville, tennessee*, 2015.
- [17] B. Asgari, R. Hadidi, J. Cao, D.-E. Shim, S.-K. Lim, and H. Kim, “Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction”, in *Ieee international symposium on high performance computer architecture (hPCA)*, IEEE, 2021.
- [18] B. Asgari, R. Hadidi, T. Krishna, H. Kim, and S. Yalamanchili, “Alrescha: A lightweight reconfigurable sparse-computation accelerator”, in *Submitted to the 26th ieee international symposium on high-performance computer architecture*, ACM, 2019, p. 233.
- [19] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, “Eridanus: Efficiently running inference of dnns using systolic arrays”, *Ieee micro*, 2019.
- [20] B. Asgari, R. Hadidi, and H. Kim, “Meissa: Multiplying matrices efficiently in a scalable systolic architecture”, in *Proceedings of international conference on computer design (iccd)*, IEEE, 2020.
- [21] B. Asgari, R. Hadidi, J. Dierberger, C. Steinichen, and H. Kim, “Copernicus: Characterizing the performance implications of compression formats used in sparse workloads”, *Arxiv preprint arxiv:2011.10932*, 2020.

- [22] A. Ginart, M. Naumov, D. Mudigere, J. Yang, and J. Zou, “Mixed dimension embeddings with application to memory-efficient recommendation systems”, *Arxiv preprint arxiv:1909.11810*, 2019.
- [23] M. Gao, G. Ayers, and C. Kozyrakis, “Practical near-data processing for in-memory analytics frameworks”, in *Parallel architecture and compilation (pact), 2015 international conference on*, IEEE, 2015, pp. 113–124.
- [24] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, “Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules”, in *High performance computer architecture (hpca), 2015 ieee 21st international symposium on*, IEEE, 2015, pp. 283–295.
- [25] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, “Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems”, in *Microarchitecture (micro), 2016 49th annual ieee/acm international symposium on*, IEEE, 2016, pp. 1–13.
- [26] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, “Tetris: Scalable and efficient neural network acceleration with 3d memory”, in *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*, ACM, 2017, pp. 751–764.
- [27] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing”, *Acm sigarch computer architecture news*, vol. 43, no. 3, pp. 105–117, 2016.
- [28] J. T. Pawlowski, “Hybrid memory cube (hmc)”, in *Hot chips 23 symposium (hcs), 2011 ieee*, IEEE, 2011, pp. 1–24.
- [29] J. Standard, “High bandwidth memory (hbm) dram”, *Jesd235*, 2013.
- [30] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, “Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory”, in *Computer architecture (isca), 2016 acm/ieee 43rd annual international symposium on*, IEEE, 2016, pp. 380–392.
- [31] A. Rodrigues, M. Gokhale, and G. Voskuilen, “Towards a scatter-gather architecture: Hardware and software issues”, in *Proceedings of the international symposium on memory systems*, 2019, pp. 261–271.
- [32] M. Gokhale, S. Lloyd, and C. Hajas, “Near memory data structure rearrangement”, in *Proceedings of the 2015 international symposium on memory systems*, 2015, pp. 283–290.

- [33] S. Lloyd and M. Gokhale, “Near memory key/value lookup acceleration”, in *Proceedings of the international symposium on memory systems*, 2017, pp. 26–33.
- [34] N. Tanabe, B. Nuttapon, H. Nakajo, Y. Ogawa, J. Kogou, M. Takata, and K. Joe, “A memory accelerator with gather functions for bandwidth-bound irregular applications”, in *Proceedings of the 1st workshop on irregular applications: Architectures and algorithms*, 2011, pp. 35–42.
- [35] Y. Kwon, Y. Lee, and M. Rhu, “Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning”, in *Micro*, 2019, pp. 740–753.
- [36] L. Ke, U. Gupta, C.-J. Wu, B. Y. Cho, M. Hempstead, B. Reagen, X. Zhang, D. Brooks, V. Chandra, U. Diril, *et al.*, “Recnmp: Accelerating personalized recommendation with near-memory processing”, *Isca*, 2020.
- [37] R. Hwang, T. Kim, Y. Kwon, and M. Rhu, “Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations”, *Isca*, 2020.
- [38] H. Kung and C. E. Leiserson, “Systolic arrays (for vlsi)”, in *Sparse matrix proceedings 1978*, Society for Industrial and Applied Mathematics, vol. 1, 1979, pp. 256–282.
- [39] S. Y. Kung, “Vlsi array processors”, *Englewood cliffs, nj, prentice hall, 1988, 685 p. research supported by the semiconductor research corp., sdio, nsf, and us navy.*, 1988.
- [40] H.-T. Kung, “Why systolic architectures?”, *Ieee computer*, vol. 15, no. 1, pp. 37–46, 1982.
- [41] H. Kung, B. McDanel, and S. Q. Zhang, “Adaptive tiling: Applying fixed-size systolic arrays to sparse convolutional neural networks”, in *2018 24th international conference on pattern recognition (icpr)*, IEEE, 2018, pp. 1006–1011.
- [42] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, “Automated systolic array architecture synthesis for high throughput cnn inference on fpgas”, in *Proceedings of the 54th annual design automation conference 2017*, ACM, 2017, p. 29.
- [43] H. Kung, B. McDanel, S. Q. Zhang, X. Dong, and C. C. Chen, “Maestro: A memory-on-logic architecture for coordinated parallel use of many systolic arrays”, in *2019 ieee 30th international conference on application-specific systems, architectures and processors (asap)*, IEEE, vol. 2160, 2019, pp. 42–50.

- [44] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, “Redeye: Analog convnet image sensor architecture for continuous mobile vision”, in *Isca’16*, ACM, 2016, pp. 255–266.
- [45] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, *et al.*, “Ese: Efficient speech recognition engine with sparse lstm on fpga”, in *Proceedings of the 2017 acm/sigda international symposium on fpga*, ACM, 2017, pp. 75–84.
- [46] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient re-configurable accelerator for deep convolutional neural networks”, *Ieee journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2017.
- [47] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, “Shidiannao: Shifting vision processing closer to the sensor”, in *Acm sigarch computer architecture news*, ACM, vol. 43, 2015, pp. 92–104.
- [48] S. Wang, D. Zhou, X. Han, and T. Yoshimura, “Chain-nn: An energy-efficient 1d chain architecture for accelerating deep convolutional neural networks”, in *Design, automation & test in europe conference & exhibition (date), 2017*, IEEE, 2017, pp. 1032–1037.
- [49] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, “In-datacenter performance analysis of a tensor processing unit”, in *Acm/ieee 44th annual international symposium on computer architecture (isca)*, IEEE, 2017, pp. 1–12.
- [50] H. Lim, V. Piuri, and E. E. Swartzlander, “A serial-parallel architecture for two-dimensional discrete cosine and inverse discrete cosine transforms”, *Ieee transactions on computers*, vol. 49, no. 12, pp. 1297–1309, 2000.
- [51] H. Kung, B. McDanel, and S. Q. Zhang, “Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization”, in *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, ACM, 2019, pp. 821–834.
- [52] R. Urquhart and D. Wood, “Systolic matrix and vector multiplication methods for signal processing”, in *Ieee proceedings of communications, radar and signal processing*, IET, vol. 131, 1984, pp. 623–631.
- [53] M. O’Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, “Fine grained dram: Energy efficient dram for extreme bandwidth systems”, in *Proceedings of the 50th annual ieee/acm international symposium on microarchitecture*, ACM, 2017, pp. 41–54.

- [54] Y. Wang, M. Zhang, and J. Yang, “Towards memory-efficient processing-in-memory architecture for convolutional neural networks”, in *Acm sigplan notices*, ACM, vol. 52, 2017, pp. 81–90.
- [55] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient re-configurable accelerator for deep convolutional neural networks”, *Ieee journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2017.
- [56] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, “Dadiannao: A machine-learning supercomputer”, in *Proceedings of the 47th annual ieee/acm international symposium on microarchitecture*, IEEE Computer Society, 2014, pp. 609–622.
- [57] J. Dean, *Machine learning for systems and systems for machine learning*, 2017.
- [58] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, *et al.*, “Scaleddeep: A scalable compute architecture for learning and evaluating deep networks”, in *Proceedings of the 44th annual international symposium on computer architecture*, ACM, 2017, pp. 13–26.
- [59] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, “Neurostream: Scalable and energy efficient deep learning with smart memory cubes”, *Arxiv preprint arxiv:1701.06420*, 2017.
- [60] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. 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*, ACM, 2017, pp. 75–84.
- [61] A. X. M. Chang and E. Culurciello, “Hardware accelerators for recurrent neural networks on fpga”, in *2017 ieee international symposium on circuits and systems (iscas)*, IEEE, 2017, pp. 1–4.
- [62] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, “Deltarnn: A power-efficient recurrent neural network accelerator”, in *Proceedings of the 2018 acm/sigda international symposium on field-programmable gate arrays*, ACM, 2018.
- [63] C. Y. Lin, N. Wong, and H. K.-H. So, “Design space exploration for sparse matrix-matrix multiplication on fpgas”, *International journal of circuit theory and applications*, vol. 41, no. 2, pp. 205–219, 2013.
- [64] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, “Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware”, in *High performance extreme computing conference (hpec), 2013 ieee*, IEEE, 2013.

- [65] B. Asgari, R. Hadidi, and H. Kim, “Ascella: Accelerating sparse computation by enabling stream accesses to memory”, in *Proceedings of automation and test in europe conference and exhibition (date)*, IEEE, 2020.
- [66] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr, “Fine-grained accelerators for sparse machine learning workloads”, in *Design automation conference (asp-dac), 2017 22nd asia and south pacific*, IEEE, 2017, pp. 635–640.
- [67] E. Nurvitadhi, A. Mishra, and D. 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*, IEEE Press, 2015, pp. 109–116.
- [68] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. M. Rush, G.-Y. Wei, and D. Brooks, “Masr: A modular accelerator for sparse rnns”, in *28th international conference on parallel architectures and compilation techniques (pact)*, IEEE, 2019, pp. 1–14.
- [69] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, “Outerspace: An outer product based sparse matrix multiplication accelerator”, in *2018 ieee international symposium on high performance computer architecture (hPCA)*, IEEE, 2018, pp. 724–736.
- [70] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, “Extensor: An accelerator for sparse tensor algebra”, in *Proceedings of the 52nd annual ieee/acm international symposium on microarchitecture*, 2019, pp. 319–333.
- [71] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N. M. Ghiassi, T. Shahroodi, J. G. Luna, and O. Mutlu, “Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations”, in *Proceedings of the 52nd annual ieee/acm international symposium on microarchitecture*, 2019, pp. 600–614.
- [72] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, “Graphr: Accelerating graph processing using reram”, in *2018 ieee international symposium on high performance computer architecture (hPCA)*, IEEE, 2018, pp. 531–543.
- [73] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek, “Enabling scientific computing on memristive accelerators”, in *2018 acm/ieee 45th annual international symposium on computer architecture (isca)*, IEEE, 2018.
- [74] Y. Huang, N. Guo, M. Seok, Y. Tsvitidis, and S. Sethumadhavan, “Analog computing in a modern context: A linear algebra accelerator case study”, *Ieee micro*, vol. 37, no. 3, 2017.

- [75] Y. Huang, N. Guo, M. Seok, Y. Tsividis, K. Mandli, and S. Sethumadhavan, “Hybrid analog-digital solution of nonlinear partial differential equations”, in *Micro*, IEEE, 2017.
- [76] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, “Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization”, in *Proceedings of the 52nd annual ieee/acm international symposium on microarchitecture*, 2019, pp. 347–358.
- [77] E. Phillips and M. Fatica, “A cuda implementation of the high performance conjugate gradient benchmark”, in *International workshop on performance modeling, benchmarking and simulation of high performance computer systems*, Springer, 2014, pp. 68–84.
- [78] K. Akbudak and C. Aykanat, “Exploiting locality in sparse matrix-matrix multiplication on many-core architectures”, *Ieee transactions on parallel and distributed systems*, vol. 28, no. 8, pp. 2258–2271, 2017.
- [79] E. Saule, K. Kaya, and Ü. V. Çatalyürek, “Performance evaluation of sparse matrix multiplication kernels on intel xeon phi”, in *International conference on parallel processing and applied mathematics*, Springer, 2013, pp. 559–570.
- [80] P. D. Sulatycke and K. Ghose, “Caching-efficient multithreaded fast multiplication of sparse matrices”, in *Parallel processing symposium, 1998. ipps/spdp 1998. proceedings of the first merged international... and symposium on parallel and distributed processing 1998*, IEEE, 1998, pp. 117–123.
- [81] S. Dalton, L. Olson, and N. Bell, “Optimizing sparse matrix-matrix multiplication for the gpu”, *Acm transactions on mathematical software (toms)*, vol. 41, no. 4, p. 25, 2015.
- [82] F. Gremse, A. Hoftler, L. O. Schwen, F. Kiessling, and U. Naumann, “Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging”, *Siam journal on scientific computing*, vol. 37, no. 1, pp. C54–C71, 2015.
- [83] W. Liu and B. Vinter, “An efficient gpu general sparse matrix-matrix multiplication for irregular data”, in *Parallel and distributed processing symposium, 2014 ieee 28th international*, IEEE, 2014, pp. 370–381.
- [84] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli, “Sparse matrix-matrix multiplication on modern architectures”, in *High performance computing (hipc), 2012 19th international conference on*, IEEE, 2012, pp. 1–10.

- [85] W. Liu and B. Vinter, “A framework for general sparse matrix–matrix multiplication on gpus and heterogeneous processors”, *Journal of parallel and distributed computing*, vol. 85, pp. 47–61, 2015.
- [86] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, “Graphicionado: A high-performance and energy-efficient accelerator for graph analytics”, in *2016 49th annual ieee/acm international symposium on microarchitecture (micro)*, IEEE, 2016, pp. 1–13.
- [87] R. W. Vuduc and H.-J. Moon, “Fast sparse matrix-vector multiplication by exploiting variable block structure”, in *International conference on high performance computing and communications*, Springer, 2005, pp. 807–816.
- [88] M. Kumar, M. Serrano, J. Moreira, P. Pattnaik, W. P. Horn, J. Jann, and G. Tanase, “Efficient implementation of scatter-gather operations for large scale graph analytics”, in *2016 ieee high performance extreme computing conference (hpec)*, IEEE, 2016, pp. 1–7.
- [89] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning structured sparsity in deep neural networks”, in *Advances in neural information processing systems*, 2016, pp. 2074–2082.
- [90] H. Li *et al.*, “Pruning filters for efficient convnets”, *Arxiv:1608.08710*, 2016.
- [91] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, “Scalpel: Customizing dnn pruning to the underlying hardware parallelism”, in *Acm sigarch computer architecture news*, ACM, vol. 45, 2017, pp. 548–560.
- [92] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, “Exploring the regularity of sparse structure in convolutional neural networks”, *Arxiv preprint arxiv:1705.08922*, 2017.
- [93] P. Molchanov *et al.*, “Pruning convolutional neural networks for resource efficient inference”, *Arxiv:1611.06440*, 2016.
- [94] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, “Towards general purpose acceleration by exploiting common data-dependence forms”, in *Proceedings of the 52nd annual ieee/acm international symposium on microarchitecture*, 2019, pp. 924–939.
- [95] R. D. Evans, L. Liu, and T. M. Aamodt, “Jpeg-act: Accelerating deep learning via transform-based lossy compression”,
- [96] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, “Gist: Efficient data encoding for deep neural network training”, in *2018 acm/ieee 45th annual*

international symposium on computer architecture (isca), IEEE, 2018, pp. 776–789.

- [97] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “Eie: Efficient inference engine on compressed deep neural network”, *Acm sigarch computer architecture news*, vol. 44, no. 3, pp. 243–254, 2016.
- [98] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, “Scnn: An accelerator for compressed-sparse convolutional neural networks”, in *Proceedings of the 44th annual international symposium on computer architecture*, ACM, 2017, pp. 27–40.
- [99] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, “Sparten: A sparse tensor accelerator for convolutional neural networks”, in *Proceedings of the 52nd annual ieee/acm international symposium on microarchitecture*, 2019, pp. 151–165.
- [100] G. E. Blelloch, M. A. Heroux, and M. Zagha, “Segmented operations for sparse matrix computation on vector multiprocessors”, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, Tech. Rep., 1993.
- [101] D. Guo, W. Gropp, and L. N. Olson, “A hybrid format for better performance of sparse matrix-vector multiplication on a gpu”, *The international journal of high performance computing applications*, vol. 30, no. 1, pp. 103–120, 2016.
- [102] A. C. I. Malossi, Y. Ineichen, C. Bekas, A. Curioni, and E. S. Quintana-Ortí, “Performance and energy-aware characterization of the sparse matrix-vector multiplication on multithreaded architectures”, in *2014 43rd international conference on parallel processing workshops*, IEEE, 2014, pp. 139–148.
- [103] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, “Understanding the performance of sparse matrix-vector multiplication”, in *16th euromicro conference on parallel, distributed and network-based processing (pdp 2008)*, IEEE, 2008, pp. 283–292.
- [104] V. Karakasis, G. Goumas, and N. Koziris, “Exploring the performance-energy trade-offs in sparse matrix-vector multiplication”, in *Workshop on emerging supercomputing technologies (west)*, vol. 11, 2011.
- [105] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix-vector multiplication on emerging multicore platforms”, in *Sc’07: Proceedings of the 2007 acm/ieee conference on supercomputing*, IEEE, 2007, pp. 1–12.

- [106] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, “Automatic selection of sparse matrix representation on gpus”, in *Proceedings of the 29th acm on international conference on supercomputing*, 2015, pp. 99–108.
- [107] D. R. Kincaid, T. C. Oppe, and D. M. Young, “Itpackv 2d user’s guide”, Texas Univ., Austin, TX (USA). Center for Numerical Analysis, Tech. Rep., 1989.
- [108] Y. Saad, “Numerical solution of large nonsymmetric eigenvalue problems”, *Computer physics communications*, vol. 53, no. 1-3, pp. 71–90, 1989.
- [109] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, “A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units”, *Siam journal on scientific computing*, vol. 36, no. 5, pp. C401–C423, 2014.
- [110] Y. Saad, *Iterative methods for sparse linear systems*. siam, 2003, vol. 82.
- [111] Z. Zhang, H. Wang, S. Han, and W. J. Dally, “Sparch: Efficient architecture for sparse matrix multiplication”, *Arxiv preprint arxiv:2002.08947*, 2020.
- [112] B. Asgari, R. Hadidi, and H. Kim, “Ascella: Accelerating sparse computation by enabling stream accesses to memory”, *Proceedings of the 23rd design, automation, and test in europe (date)*, 2020.
- [113] ASU and ARM, *Arizona State Predictive PDK*, <http://asap.asu.edu/asap/>, [Online; accessed April-2020], 2020.
- [114] R. Wang, B. Fu, G. Fu, and M. Wang, “Deep & cross network for ad click predictions”, in *Proceedings of the adkdd’17*, 2017, pp. 1–7.
- [115] *Criteo AI Labs Ad Kaggle*, <https://www.kaggle.com/c/criteo-display-ad-challenge>, [Online; accessed April-2020], 2020.
- [116] *Criteo AI Labs Ad Terabyte*, <https://labs.criteo.com/2013/12/download-terabyte-click-logs/>, [Online; accessed April-2020], 2020.
- [117] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection”, *Acm transactions on mathematical software (toms)*, vol. 38, no. 1, p. 1, 2011.
- [118] Micron, “Calculating memory power for ddr4 sdram”, in, 2017.
- [119] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A system for large-scale graph processing”, in *Proceedings of the 2010 acm sigmod international conference on management of data*, ACM, 2010, pp. 135–146.

- [120] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, “Gunrock: A high-performance graph processing library on the gpu”, in *Sigplan notices*, ACM, vol. 51, 2016, p. 11.
- [121] X. Zhu, W. Han, and W. Chen, “Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning”, in *Usenix act*, 2015, pp. 375–386.
- [122] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, “Cusha: Vertex-centric graph processing on gpus”, in *Hpdac*, ACM, 2014.
- [123] B. Asgari, S. Mukhopadhyay, and S. Yalamanchili, “Mahasim: Machine-learning hardware acceleration using a software-defined intelligent memory system”, *Journal of signal processing systems, special issue on embedded machine learning*, 2019.
- [124] S. Anwar *et al.*, “Structured pruning of deep cnns”, *Jetc*, vol. 13, no. 3, p. 32, 2017.
- [125] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, “In-datacenter performance analysis of a tensor processing unit”, in *Computer architecture (isca), 2017 acm/ieee 44th annual international symposium on*, IEEE, 2017, pp. 1–12.