HARDWARE-SOFTWARE TECHNIQUES FOR ACCELERATING SPARSE COMPUTATION

By

MOHAMMADREZA SOLTANIYEH

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Santosh Nagarakatte

And approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2022

ABSTRACT OF THE DISSERTATION

Hardware-Software Techniques for Accelerating Sparse Computation

by **Mohammadreza Soltaniyeh**

Dissertation Director: Prof. Santosh Nagarakatte

Linear algebra kernels are widely used in various fields such as machine learning, data science, physical science, and graph analysis. Many of these applications work with sparse data (*i.e.*, only a small fraction of data is non-zero). Sparse data are often stored in a compressed format (ie, sparse format) that stores only the non-zero elements with additional metadata to identify where the non-zero elements are located. Using compressed formats eliminates the need to store and process zeros, making the storage and computation of sparse kernels more efficient.

General purpose architectures, such as CPUs and GPUs, are not able to deliver the same performance for sparse linear algebra kernels as they do for dense versions. First, accessing non-zero elements in sparse format introduces many indirect and irregular memory accesses incompatible with SIMD and caching mechanisms used by CPUs and GPUs. In addition, Dennard scaling is obsolete and Moore's law is slowing down, ending the era in which general-purpose architectures become faster and more energy efficient transparently. This has led to a plethora of research into developing specialized hardware, such as FPGAs and ASICs to improve the performance and energy efficiency of these sparse kernels. A key strategy for the specialized hardware is to customize the sparse format (i.e., storage) according to the operation memory access pattern, the pattern of non-zero elements in the input (*i.e.*, sparsity pattern), and the underlying hardware structures. This approach is effec-

tive if the operations and input sparsity patterns do not change. However, applications often perform various operations on sparse data. Additionally, the sparse inputs may frequently change for each execution, and each input may have a different sparsity pattern. When this happens, the performance of specialized hardware degrades because a reformatting step is required to convert the data into a format that is compatible with the hardware. The data reformatting can be expensive when it cannot be overlapped with the computation on the hardware or amortized over multiple application executions with the same input data.

This dissertation presents a few hardware-software techniques that enhance the performance and energy efficiency of some of the most important sparse problems, including sparse matrix-vector multiplication (SpMV), sparse general matrix-matrix multiplication (SpGEMM), and sparse convolutional neural networks (CNNs). The key insight of our method is to use the software to reformat the sparse data into a hardware-friendly format, allowing the hardware to perform the computation with a high degree of parallelism. The software improves design flexibility by supporting multiple sparse formats, and the hardware improves performance and energy efficiency. We applied these hardware-software techniques to SpMV, SpGEMM, and sparse CNNs. These problems have different characteristics, such as different input densities and distinct input sparsity pattern features. The contribution of this dissertation can be summarized as follows. First, we present a synergistic CPU-FPGA system to accelerate SpMV and SpGEMM kernels. In our proposed design, the CPU reorganizes sparse data into a format suitable for the FPGA, and the FPGA computes with high parallelism using the preprocessed data. We develop an intermediate representation that allows the software to communicate regularized data and scheduling decisions to the FPGA. Besides, most of the CPU and FPGA execution are overlapped. Our approach can effectively handle sparse kernels with low input densities and sparsity patterns varying for each sparse input. Our end-to-end full system evaluation of the REAP prototype using Alveo-U200 FPGA has up to 3.4$\times$ and 1.3$\times$ speedup over the highly optimized Intel MKL library on a multicore CPU for SpMV and SpGEMM for widely-used

sparse formats. Our results show that REAP achieves both high frequency and promising speedup compared to state-of-the-art FPGA accelerators for SpMV and SpGEMM while supporting various sparse formats and precision configurations. Second, we present a hardware accelerator for sparse CNN inference tasks. We formulate the convolution operation as general matrix-matrix multiplication (GEMM) using an image to column (IM2COL) transformation. With a dynamically reconfigurable GEMM and a novel IM2COL hardware unit, our design can support various layers in CNNs with high performance. Besides, our design exploits sparsity in both weights and feature maps. We use the software to perform group-wise pruning followed by a preprocessing step that puts the pruned weights into our hardware-friendly sparse format for efficient and high performance computation. We evaluated our accelerator using a cycle-level simulator and an HLS implementation realized on an Alveo FPGA board. Our ASIC design is on average 2.16×, 1.74×, and 1.63× faster than Gemmini, Eyeriss, and Sparse-PE, which are prior hardware accelerators for dense and sparse CNNs, respectively. Besides, our hardware accelerator is also 78×, and 12× more energy-efficient when compared to CPU and GPU implementations, respectively.

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisor, Prof. Santosh Nagarakatte, for his support and guidance throughout my Ph.D. journey. Santosh spent countless hours teaching me how to think rigorously and critically, present ideas, and clearly express ideas in writing. His influence on how to think, present, and write will always remain with me.

I also extend my deepest gratitude to Prof. Richard P. Martin. His collaboration and guidance have been invaluable throughout my Ph.D.

I would also like to thank the other members of my Ph.D. committee, Prof. Yipeng Huang and Prof. Joe Devietti, for their thoughtful feedback about my research that has helped shape this dissertation.

Additionally, I would like to thank the Memory Solution team at Samsung Semiconductor, where I did two internships during my Ph.D. I particularly want to thank my mentors and manager, Veronica Lagrange Moutinho dos Reis, Matt Bryson, and Xuebin Yao, from whom I learned a lot.

I feel fortunate to have had the opportunity to learn from and interact with faculty members of the Computer Science Department at Rutgers University, particularly Abhishek Bhattacharjee, Srinivas Narayana, Sudarsun Kannan, Thu Nguyen, and all the other faculty that I have been able to learn from. I am especially thankful to Abhishek Bhattacharjee for providing valuable feedback about the initial directions of my research. I would also like to thank the administrative and technical staff at the Computer Science Department at Rutgers University for their support.

At Rutgers, I made some great friends who made this journey enjoyable. I want to thank my lab mates at RAPL group, David Menendez, Adarsh Yoga, Nader Boushehriinejad, Jay Lim, Sangeeta Chowdhary, Harishankar Vishwanathan, and Matan Shachnai, for their support. Beyond my lab, I was fortunate to meet some great friends, Guilherme, Zi Yan, Binh

Pham, Jan Vesely, Georgiana, and Jaewoo. They brought fun and a sense of motivation.

Last but not least, I would like to thank all my family for their unconditional love and support throughout the years. I would also like to thank Mahdi, Alireza, Kathryn, Shahab, Mahyar, Vahid, Shiva, and many other friends with whom I had the opportunity to have fun times.

To my family

TABLE OF CONTENTS

Abstract ii

Acknowledgments v

List of Tables xii

List of Figures xiii

Chapter 1: Introduction 1

1.1 Dissertation Statement 4

1.1.1 Synergistic CPU-FPGA Acceleration for SpMV and SpGEMM . . . 5

1.1.2 An Accelerator for Sparse Convolutional Neural Networks Leveraging Systolic General Matrix-Matrix Multiplication 8

1.1.3 A End-to-end FPGA Prototype of SPOTS for Sparse CNNs 11

1.2 Papers Related to this Dissertation 12

1.3 Dissertation Organization 13

Chapter 2: Sparse Linear Algebra Acceleration For Multiple Formats Using A CPU-FPGA System 14

2.1 Overview of Our CPU-FPGA system for Accelerating Sparse Linear Algebra Kernels 15

2.1.1 Contributions 17

2.2 Background on Sparse Formats and Sparse Linear Algebra Kernels 17

2.2.1 A Background on Sparse Formats 18

2.2.2 A Background on SpMV and SpGEMM kernels 20

2.3 Synergistic CPU-FPGA Acceleration 23

2.3.1 Instantiation of REAP to Accelerate SpMV 28

2.3.2 Instantiation of REAP to Accelerate SPGEMM 32

2.4 Experimental Methodology of REAP 39

2.5 Experimental Evaluation of REAP for SpMV and SpGEMM 42

2.6 Related Work on Accelerators for SpMV and SpGEMM Kernels 49

2.7 Summary 52

Chapter 3: An ASIC Accelerator for Sparse Convolutional Neural Networks . . 53

3.1 Overview of SPOTS 54

3.1.1 Novelties of SPOTS 57

3.2 Background on CNNs, IM2COL, and Sparsity-Awareness in CNNs 57

3.2.1 Convolution Neural Networks 57

3.2.2 Transforming Convolution to General Matrix-Matrix Multiplication 59

3.2.3 Sparsity-Awareness in CNNs 61

3.3 Motivation for a GEMM-based Formulation of Convolution and Sparsity-Awareness Design in SPOTS 62

3.3.1 The Inefficiencies of Mapping Convolution Operation to Processing Elements 62

3.3.2 Benefits and Challenges of Convolution with IM2COL 63

3.3.3 Drawbacks of Prior Sparsity-aware Designs 64

3.4 SPOTS: Our Hardware Accelerator for Sparse CNNs 65
3.4.1 The IM2COL Unit 67
3.4.2 The GEMM Unit 70
3.4.3 Handling Sparsity in CNNs 73
3.4.4 Handling Various CNN Layers/Shapes 77
3.4.5 Strategies to Improve Load Balance in SPOTS 79
3.5 Related work 80
3.6 Summary 84

Chapter 4: An FPGA Accelerator for Sparse Convolutional Neural Networks . . 85
4.1 Background on High Level Synthesis for FPGAs 86
4.1.1 Building FPGA Designs with High Level Synthesis 86
4.1.2 HLS Optimizations 86
4.2 The Architecture of Our FPGA Accelerator for Sparse CNNs 91
4.2.1 The IM2COL Unit 91
4.2.2 The GEMM Unit 96
4.2.3 Sparsity-aware Design 99
4.2.4 Design Scalibility and Handling Various CNN Layers 102
4.3 Related Work on FPGA Accelerators for CNNs 104
4.4 Summary 106

Chapter 5: Experimental Evaluation of our ASIC and FPGA Accelerators for Sparse Convolutional Neural Networks 107
5.1 Experimental Methodology 107

5.2 Experimental Evaluation of SPOTS 112

5.2.1 Comparing the Speedup of SPOTS ASIC and FPGA Prototypes with CPUs and GPUs 113

5.2.2 Comparing the Energy Efficiency of SPOTS ASIC and FPGA Prototypes with CPUs and GPUs 114

5.2.3 Comparing the Speedup of SPOTS ASIC Prototypes with Other ASIC Designs 114

5.2.4 Performance Sensitivity to Different Layers' Shapes 119

5.2.5 Performance and Energy Characterization of IM2COL Unit 121

5.2.6 Load Imbalance in SPOTS 122

Chapter 6: Conclusion and Future Directions 123

6.1 Dissertation Summary 123

6.2 Directions for Future Work 125

6.2.1 Extending REAP to other Sparse Linear Algebra Kernels 125

6.2.2 Synthesizing Hardware Accelerators for Sparse Problems 126

6.2.3 Exploring New Programming Languages for Sparse Kernels on FPGAs 126

LIST OF TABLES

2.1 The CPU, and FPGA configurations. 39

2.2 FPGA resource utilization for SpMV and SpGEMM designs. 39

2.3 List of matrices from SparseSuite [27] used in our evaluation. 41

2.4 REAP speedup compared to a 16-core Intel MKL for SpMV and SpGEMM for three sparse formats and different input precisions. 42

2.5 Qualitative comparison of REAP with prior work that use FPGAs to accelerate sparse linear algebra kernels. 51

3.1 Qualitative comparison of SPOTS with prior work. 82

5.1 SPOTS ASIC design parameters and area. 108

5.2 The CPU, GPU and FPGA configurations for SPOTS evaluation. 108

5.3 The FPGA configurations for SPOTS evaluation. 109

5.4 FPGA resource utilization and operating frequency for two versions of SPOTS on Alveo U200. 109

5.5 Network characteristics, the top1, and top5 result accuracy, and the overall sparsity for the original (with no pruning), random pruning, and our structured pruning using the imagenet dataset. 110

5.6 Comparing the performance and efficiency of SPOTS ASIC design with different ASIC designs for AlexNet and VGGNet. 116

LIST OF FIGURES

2.1 An illustration of different sparse format representations, including COO, CSR, CSC, DIA, ELL, and RLC. 19

2.2 An illustration of four different formulations to perform general matrix-matrix multiplication. 21

2.3 The floorplan of XCU200 FPGA. 25

2.4 An illustration of the overlapping of the tasks performed by the CPU and FPG tasks in REAP. 27

2.5 An illustration of RIR bundle generation for SpMV. 30

2.6 An illustration of the FPGA design for SpMV. 31

2.7 An illustration of RIR bundle generation for SpGEMM. 34

2.8 Overall architecture of the FPGA dsign for SpGEMM. 34

2.9 An illustration of the multiply unit in action for SpGEMM design. 35

2.10 Speedup of REAP compared to 16-core Intel MKL for the SpMV and SpGEMM kernels with single-precision inputs stored in the CSR format. . . 43

2.11 Comparing the end-to-end execution time for two versions of ELL and DIA formats. 44

2.12 Comparing REAP's speedup with the recent FPGA accelerators for SpMV and SpGEMM. 45

2.13 REAP execution breakdown. 47

2.14 Studying the impact of load imbalance on the FPGA execution time for SpGEMM and SpMV. 49

3.1 An illustration of a convolution layer along with its inputs. 58

3.2 An illustration of an FC layer and its matrix form. 59

3.3 An illustration of transforming a convolution layer to a GEMM computation. 60

3.4 A comparison of various pruning methods for CNNs. 61

3.5 An illustration showing the percentage of the total execution time spent in the IM2COL transformation for various convolution layers from AlexNet, VGGNet, and GoogleNet for a CPU system. 63

3.6 The overall architecture of our accelerator with the IM2COL unit and a systolic array-based GEMM unit. 67

3.7 An illustration of patch generation using the PUs in the IM2COL unit. . . . 68

3.8 An illustration of the GEMM unit in our accelerator. 71

3.9 A comparison of our proposed custom sparse format with other state-of-the-art sparse formats. 75

3.10 An illustration of our proposed sparse format to store pruned weights and our sparsity-aware architecture. 76

3.11 An illustration of different configurations of the systolic array GEMM unit. 77

4.1 An illustration of loop pipelining with HLS. 87

4.2 An illustration of loop unrolling with HLS. 88

4.3 An illustration of dataflow pragma in HLS. 89

4.4 An illustration of array partitioning with HLS. 90

4.5 A high-level overview of IM2COL and patch units. 92

4.6 An illustration of the IM2COL unit in our FPGA design for sparse CNNs. . 93

4.7 The architecture of the GEMM unit in our FPGA deisgn for sparse CNNs. . 98

4.8 An illustration of our sparse format in comparison to the CSC format. . . . 101

4.9 A demonstration of our design's flexibility in supporting CNN layers with a variety of filters. 103

5.1 An illustration showing the sparsity ratio in the filters and input feature maps for AlexNet, VGGNet, ResNet, and GoogleNet. 111

5.2 Evaluating the ASIC and FPGA prototypes of SPOTS in performance and energy efficiency compared to the CPU and GPU implementations for sparse CNNs. 113

5.3 Evaluating the ASIC prototype of SPOTS in performance compared to Sparse-PE, Eyeriss, and Gemmini for four CNNs: AlexNet, VGGNet, ResNet, and GoogleNet. 115

5.4 Evaluating the MAC utilization for various GEMM configurations for SPOTS, ASIC and FPGA prototypes. 117

5.5 Performance characterization of the IM2COL unit in SPOTS. 118

5.6 The load imbalance percentage in the pruned weights for AlexNet, VGGNet, ResNet, and GoogleNet. 120

CHAPTER 1
INTRODUCTION

Many problems from various domains, including simulating physical body dynamics [58], multigrid methods [41], network routing[153], graph processing [65], and neural networks [57] can be expressed in terms of operations on vectors and matrices. For an important class of these problems, most of the elements in the matrices or vectors are zeros. When a substantial fraction of the elements in the data is zero, the data is called sparse. A dense representation of sparse data consumes more storage and involves many unnecessary operations on zeros. Hence, it is efficient to store sparse data using a compressed format (*i.e.*, sparse format) where only the non-zero elements are stored. Sparse formats offer two advantages. First, a sparse format can reduce the total storage by orders of magnitude compared to a dense representation. In addition, operations on zeros can be skipped to reduce the overall computation.

Sparse formats store only the non-zero elements and use various techniques to encode the positions of the non-zero elements within the matrix [15, 26, 72, 91, 104, 131]. In most standard sparse formats, determining the position of the non-zero elements requires a series of indirect memory accesses that introduce irregular memory accesses. For example, to access an element $A[i, j]$ in the Compressed Sparse Row (CSR) [104] format, one needs to consult the row pointer to obtain the location where the *row i* begins (*i.e.*, $row\_pointer[i]$), search the column indices until the beginning of the next row to check if the item is present (*i.e.*, search from $col[row\_pointer[i]]$ to $col[row\_pointer[i + 1]]$), and then subsequently access the data element at the matched index. In addition, skipping the operation involving zeros requires extra work to determine if the non-zero elements match before performing the operation. The overhead of accessing the indices into the sparse structure, not including the matching cost, can exceed the work of the mathematical operations by 2×-5× [59, 61].

In addition to the steps involved in extracting and matching the non-zero elements in sparse data, there are other factors that make sparse computation more complex than the dense version. First, the *density* of the non-zeros in the sparse input can vary wildly for various applications. The density ratio is defined as the number of non-zero elements divided by the total number of elements in the dense representation. For example, in a neural network, the density varies across the layers anywhere from 20% to 80%. By contrast, in graph processing, the input densities are as low as $10^{-6}\%$. The density of non-zeros can impact the irregularity of memory accesses. Sparse data with a low-density exhibit more random access to memory. As a second factor, the distribution of non-zeros in the data referred to as the *sparsity pattern* affects the complexity of memory accesses in a sparse problem. It is easier to store and compute sparse data with a regular sparsity pattern (when all the non-zero values are around the diagonal). Besides, the sparsity pattern can be adjusted for some applications. For example, pruning is a method used to remove redundant weights in neural networks after training. Eliminating the weights creates zeros in the weights. Different pruning methods result in different sparsity patterns. A random pruning approach can leave zeros at any location in the input. Alternatively, a structured pruning technique leaves zeros in well-defined locations. Thus, the sparsity patterns can be controlled by choosing different pruning methods. In many other applications, the sparsity pattern cannot be altered. Furthermore, the sparse inputs in some applications can often change, with each input having a different pattern of sparsity. In contrast, the sparsity pattern of input (i.e., pruned weights) remains unchanged throughout the inference process in neural network inference tasks.

The difference in delivered Floating Point Operations per Second (FLOPs) between dense and sparse kernels can be significant depending on the density of the sparse input and the sparsity pattern [128, 142]. There have been numerous efforts to build high-performance software libraries and frameworks for sparse kernels to reduce the gap between the performance of sparse and dense versions [1, 2, 3, 76, 123, 130]. CPUs and

GPUs use a Single Instruction Multiple Data (SIMD) model to perform more useful work per instruction. Additionally, they rely on caching techniques to maximize memory performance by utilizing temporal and spatial data localities. SIMD and caching techniques are effective for most dense computations, but they are not as effective for sparse problems. Using sparse formats introduces many indirect and irregular memory accesses, which limits the benefits of caching and the SIMD model. As a result, using the same formulations and optimizations as the dense version for the sparse version results in poor performance.

For many years, Moore's Law [92] and Dennard scaling [32] helped general-purpose processors become faster and more energy efficient transparently. Dennard scaling is obsolete, and Moore's law has slowed down and is expected to end in the coming years. In response, there has been an increased interest in designing *specialized hardware*, such as ASICs and FPGAs, for various applications, including sparse problems. Specialization can be applied at different levels for sparse computation. At the algorithmic level, new algorithms and methods can be developed to perform more optimally for sparse scenarios [15, 46, 137]. For sparse data storage, the sparse formats can be customized based on the operations' memory access pattern and sparsity pattern of the inputs [38, 61, 97, 120]. Lastly, custom hardware can be designed to perform operations such as extracting the non-zero elements or matching the non-zeros more efficiently [7, 42, 52, 61]. These customizations aim to improve sparse computation's performance and energy efficiency by extracting parallelism at a finer level and optimizing the memory hierarchy to accommodate irregular memory accesses arising from sparse problems. Sparse Matrix-Vector Multiplication (SpMV), Sparse General Matrix-Matrix Multiplication (SpGEMM), and sparse neural networks are among the most studied sparse problems for these specialized hardware.

The custom hardware accelerators improve sparse problems' performance and energy efficiency at the expense of generality. Often, sparse data is subject to multiple operations, and each operation may require the data to be stored in a different format for optimal performance. For example, the COO format makes it easier to import data into a sparse

matrix since it is efficient for appending non-zeros [10]. However, SpMV performs twice as fast if the sparse input is stored in CSR instead of COO format [127]. Thus, sparse data are usually stored in standard sparse formats that are suitable for various operations. The discrepancy between the sparse format customized for an application and the format used to store the data on the memory requires a preliminary reformatting step. The data reformatting can be expensive if it cannot be amortized over multiple application executions with the same input data.

1.1 Dissertation Statement

This dissertation presents a number of hardware-software techniques to improve the performance and energy efficiency of sparse linear algebra kernels, including SpMV and SpGEMM and sparse convolutional neural networks. Our general strategy is to use software methods to reformat the sparse data into a hardware-friendly format that allows the hardware to perform the computation with a high degree of parallelism. The software improves design flexibility to support multiple sparse formats, and the hardware improves performance and energy efficiency. We develop an intermediate representation that allows the software to communicate regularized data and scheduling decisions to the hardware. Besides, most of the software and hardware execution are overlapped. We applied these hardware-software techniques to three sparse problems: sparse matrix-vector multiplication, sparse general matrix-vector multiplication, and sparse convolutional neural network. Different characteristics of these three applications raise different questions, and we have answered some of them in the following contributions:

1. A CPU-FPGA system to improve the performance of SpMV and SpGEMM kernels in comparison to CPU-only and FPGA-only designs, while supporting multiple sparse formats and data precisions.

2. An intermediate representation that enables the CPU to communicate the regularized

sparse data and the scheduling decisions to the FPGA and allow them to compute in parallel.

3. An ASIC and an FPGA accelerator to improve the performance and energy efficiency of sparse CNN inference task by building a hardware unit to perform IM2COL coupled with a reconfigurable systolic array-based general matrix-matrix multiplication unit.

4. A sparsity-aware design for sparse CNNs that minimizes storage while skipping the computation involving zeros without requiring complex hardware.

1.1.1 Synergistic CPU-FPGA Acceleration for SpMV and SpGEMM

Sparse linear algebra kernels such as SpMV and SpGEMM are the key components of many applications from various domains [58, 65, 153]. Many of these applications have input with very low densities (*e.g.*, below 1%). With sparse kernels, the challenge is to extract enough parallelism to improve performance while reducing storage and avoiding computation on zeros.

FPGAs offer great performance for compute-intensive applications with their programmability and massive parallelism. FPGAs achieve finer-grained parallelism than general-purpose architectures because they can customize memory hierarchy and computation engines for specific applications. However, most sparse problems are memory-bound due to the irregular and random nature of their memory accesses. Sparse problems suffer from low external memory bandwidth. The limited on-chip memory of FPGAs cannot fully compensate for sparse kernels' low external memory bandwidth. Hence, FPGAs cannot extract enough parallelism to compensate for their lower frequency to offer significant performance gains over general-purpose architectures for sparse problems.

These challenges can be addressed in several ways for FPGAs. One common technique is to build a sparse format customized for the FPGA's memory hierarchy and its computa-

tion engines [35, 119]. Another effective method is to use a software scheduler that analyzes the sparsity pattern and schedules non-zero input pairs to an FPGA microarchitecture to increase the resource utilization [54]. Both of these approaches can improve the performance of FPGA designs for sparse kernels by increasing their parallelism, which compensates for the lower frequency of FPGAs compared to CPUs. However, both methods require an expensive preprocessing step that involves software and cannot be overlapped with hardware computation.

In Chapter 2, we introduce REAP, a CPU-FPGA system for sparse linear algebra that incorporates both CPU and FPGA. REAP aims to accomplish three goals. Our first goal is to support a wide range of standard sparse formats. Hardware accelerators often require data to be available in their customized format and thus do not support standard sparse formats directly [35, 38, 54, 119, 120]. Due to the disparity between the compressed format used by the accelerator and the format used to store data in memory, these approaches require a reformatting step to convert the sparse data into the hardware supported format. Data reformatting is only justified if it can be amortized over multiple program executions with the same sparse input. Further, we want the data reformatting step in software to be overlapped with the FPGA computation. Second, we want to make our design efficient and optimized for high precision inputs (e.g., Float) as well as low precision inputs (e.g., Int8), depending on the application requirement. Relaxing the data precision of sparse kernels can lead to significant performance gains [6]. Therefore, our design can be adapted to different data precisions to further improve performance. Our third objective is to propose a generic design that can be applied to a variety of sparse linear algebra kernels. There are several hardware accelerators designed for a single sparse kernel [38, 54, 97, 118, 120]. In contrast, our method is generic and can be applied to a variety of sparse kernels. The generality of our approach is demonstrated by building designs for two sparse kernels with different complexities, namely, sparse matrix-vector multiplication (SpMV) and sparse general matrix-matrix multiplication (SpGEMM).

REAP combines the strengths of both the CPU and the FPGA. To maintain high PE utilization, the CPU reorganizes sparse data into a format suitable for the FPGA. CPUs are good at manipulating small-scale, unpredictable memory access patterns as they have high clock rates and multiple levels of cache, while FPGAs with an application-specific routing and an abundant number of DSP units and on-chip memory are suitable targets to perform numerical computation on the reformatted streams of data. In the reorganization task, the CPU identifies the elements that match and schedules the computation for the FPGA. The FPGA design consists of replicated pipelines with a large number of multipliers. To convey reorganized data and scheduling information from the CPU to the FPGA, we developed a new intermediate representation. The FPGA reads the regularized data in the intermediate format. We organize the entire computation in stages to facilitate parallel execution on the CPU and FPGA. When the CPU preprocesses the data for step k, the FPGA performs the computation corresponding to step $k-1$. By overlapping CPU and FPGA execution, performance is further improved. An important feature of REAP is that it requires only a change to the software on the CPU to adapt to a new sparse format. Thus, we can support multiple sparse formats with a single FPGA design. In addition, our intermediate representation can be adjusted to different sparsity patterns, input sizes, and data precisions.

We have synthesized a prototype of REAP for SpMV and SpGEMM for commonly used sparse formats using the Xilinx HLS toolchain on an Alveo-U200 FPGA board. Our prototype supports CSR, ELL, and DIA sparse formats and a wide variety of precision (float, integers of various bit-widths). Our end-to-end evaluation of the system on large matrices with various sparsity patterns shows that REAP, on average, exhibits up to $3.4\times$ and $1.3\times$ speedup over multi-core versions of highly optimized Intel Math Kernel Libraries on a CPU for SpMV and SpGEMM, respectively. Finally, REAP achieves high frequency and delivers promising speedup compared to state-of-the-art FPGA accelerators for SpMV and SpGEMM while offering flexibility in supporting various sparse formats.

Many sparse problems, such as the sparse neural network, can be reduced to SpMV

and SpGEMM computations. The design presented in Chapter 2 targets sparse problems with very low input densities (below 1%) and varying inputs. There is a need for a different hardware design to improve the performance of sparse problems with higher input densities or applications with sparse inputs that do not change frequently. A sparse convolutional neural network is one such application. Next, we describe how we apply our software-hardware approach to sparse convolutional neural networks.

1.1.2 An Accelerator for Sparse Convolutional Neural Networks Leveraging Systolic General Matrix-Matrix Multiplication

Neural networks are extensively used to solve complex problems in numerous domains such as video processing [68], speech recognition [23], and natural language processing [50, 113]. Convolutional neural network (CNN) is one of the most widely used neural networks. CNNs can have multiple types of layers, including convolution layers, fully connected layers, and pooling layers, with most of the computation performed in the convolution layers. Each CNN layer has multiple features, such as number of filters, kernel size, stride size, and channel size. The computation of each layer produces a higher-level abstraction of the input data, called a feature map.

To achieve higher accuracy, neural networks often include many layers. Neural networks with many layers present both performance and energy efficiency challenges to the underlying processing hardware. Luckily, most neural networks have significant redundancy that can be pruned during training without substantially reducing accuracy [48, 49]. The elimination of redundant weights will result in a network with a large number of zero values, which can potentially reduce inference computation and storage requirements. In addition to zeros in the weights, zeros also appear on the feature map while performing the inference task.

The prevalence of CNNs and the complexity of exploiting sparsity in CNNs has led to a large body of work on building hardware accelerators for sparse CNNs [8, 19, 21, 30, 36,

40, 48, 98, 102, 103, 115, 138, 145]. Given the variety of layers with different features in a CNN, it is difficult to design a hardware accelerator that performs optimally for all types of layers. Thus, some accelerators only support a few types of layer in CNNs [48, 139] or they perform optimally only for certain types of layer [21, 98]. Supporting sparse inputs makes designing CNN hardware challenging. A sparsity-aware design exploits the sparsity of one input (i.e., one-side sparsity) [8, 48, 145], or both inputs (weights and feature maps) [30, 98, 102] to enhance the performance and the energy efficiency of the sparse CNN computation. There are many ways to utilize sparsity in CNNs. First, the sparsity can be used to reduce energy consumption by gating operations involving zeros [21]. Additionally, the computation involving zeros can be skipped, improving the performance in addition to the energy consumption [30, 42, 98].

In Chapter 3 we present the details of our hardware accelerator that implements the convolution layer as a single large general matrix-matrix multiplication (GEMM) operation using an image to column transformation (IM2COL). Using GEMM as the core computation unit allows us to support a wide range of CNN layers. We discovered that the IM2COL operation accounts for a sizable fraction of the execution time (29% of the total time). Further, IM2COL performs many redundant memory accesses, contributing to the overall energy consumption. Additionally, doing the IM2COL operation in software instead of hardware prevents fine-grained pipelining of the IM2COL transformation and matrix-multiplication operations. Hence, we design both IM2COL and and GEMM units in hardware. We call our sparse convolutional network accelerator SPOTS. The three key innovations in SPOTS are: (1) a novel IM2COL unit that allows us to pipeline GEMM and IM2COL computations to improve performance, (2) a dynamically reconfigurable GEMM unit with the capability to adapt to different CNN layers and shapes, and (3) sparsity awareness that allows the design to support sparsity in both the feature map and filters. Combining these techniques increases CNN performance and energy efficiency over prior accelerators for CNNs.

A dedicated hardware* IM2COL *unit in SPOTS. We propose a dedicated hardware IM2COL unit that operates in parallel with the hardware GEMM unit. With the specialized IM2COL unit, we can reduce redundant accesses, resulting in faster inference speed and lower energy consumption. A novel aspect of the IM2COL unit in SPOTS is that it has a collection of patch units (PUs) that streams the input only once, performs data reorganization, creates multiple patches in parallel, and eliminates redundant accesses. Each patch unit in the IM2COL unit has three local buffers that identifies overlapped elements between patches and avoids expensive DRAM accesses. These patches are subsequently fed into a systolic array-based GEMM unit.

SPOTS is sparsity-aware. SPOTS efficiently handles zeros in both inputs: weights and the input feature map. SPOTS uses sparsity to skip data transfer and computation for zeros. By using a group-wise pruning technique with a new sparse format, we can reduce the storage requirements while still allowing high-speed access to the weights necessary to keep the PEs active. Further, SPOTS tags and skips blocks of zeros in the result of the IM2COL unit and weights before entering the systolic array, saving computation cycles and memory transfers. Finally, SPOTS can also prevent load imbalances caused by an uneven distribution of zeros in the inputs by skipping the zero blocks for all PEs.

A dynamically reconfigurable GEMM unit in SPOTS. The GEMM unit in SPOTS can be configured as multiple GEMM units with square-shaped systolic arrays with processing elements (PEs) or a single tall-thin unit. The tall-thin shape better balances the memory bandwidth requirement of the GEMM unit and the IM2COL throughput, which allows efficient pipelining of operations between the PEs performing the matrix multiplication and the PUs executing the IM2COL reorganization. The dynamic reconfigurability of the GEMM units enables SPOTS to achieve high PE utilization for various kinds of convolutional layers that differ in number of filters, kernel size, stride size, and input dimensions. In addition to the convolution and fully connected layers, SPOTS supports pooling layers with a minor

enhancement to the IM2COL unit.

Improvement in performance and energy efficiency with SPOTS. The techniques in SPOTS improve CNN performance and energy efficiency over prior accelerators. We evaluate our design for four popular CNNs, AlexNet, VGGNet, ResNet, and GoogleNet, which features a diverse set of convolution layers with different memory and computation requirements. We compare the performance and energy efficiency of SPOTS with other state-of-the-art hardware accelerators for CNNs. Our results show that SPOTS is on average 2.16×, 1.74×, and 1.63× faster than Gemmini [40], Eyeriss [21] and, Sparse-PE [102] respectively. SPOTS is also 78× and 12× more energy efficient when compared to CPU and GPU systems, respectively. In addition, we demonstrate that SPOTS can achieve high PE utilization under different CNN shapes.

1.1.3 A End-to-end FPGA Prototype of SPOTS for Sparse CNNs

Designing and manufacturing ASICs can take a long time and cost thousands of dollars. FPGAs are an alternative solution for building custom hardware. FPGA reconfigurable substrates reduce non-recurring engineering (NRE) costs and can be reprogrammed for different applications. Despite these advantages, FPGAs have some disadvantages when compared to ASICs. FPGAs operate at a lower frequency than ASICs partially due to their reconfigurability features. Therefore, FPGAs are slower than ASICs unless they can take advantage of more parallelism to compensate for their lower frequency.

FPGAs are also constrained by the number of available resources, such as the on-chip memory. Most FPGAs have very limited on-chip memory resources compared to CPUs' multi-level caches and GPUs' on-chip memory. In Chapter 4 we present our end-to-end FPGA design for sparse convolution neural network based on our design in Chapter 3. To build our end-to-end FPGA accelerator, we borrowed some of the main design strategies from our ASIC design, such as offloading the IM2COL and GEMM computations to the

hardware and our sparsity-awareness design. However, we revisited some aspects of the design, including the design for the IM2COL and the GEMM units to better suit FPGA. There are two main advances in our FPGA design. First, our design is scalable to different FPGAs with different resource constraints. Second, unlike many prior FPGA designs, our design uses sparsity in feature maps and weight inputs without introducing additional complexity. We describe how we used the high level synthesis (HLS) tools to build an end-to-end prototype of our design. Our end-to-end evaluation on Alveo U200 FPGA shows that by exploiting sparsity in both inputs and by overlapping the IM2COL and GEMM computation, our design can achieve better or similar CPU performance with a frequency $20\times$ less than the CPU. Besides, our FPGA solution is more energy-efficient than the CPU and GPU implementations.

1.2 Papers Related to this Dissertation

This dissertation presents the ideas and techniques presented in the following publications written in collaboration with my advisor Santosh Nagarakatte and other collaborators, Richard P. Martin from Rutgers University, Veronica Lagrange, Matt Bryson, and Xuebin Yao from Samsung Semiconductor's memory solutions lab.

1. "Synergistic CPU-FPGA Acceleration of Sparse Linear Algebra," [115], which presents a software/hardware technique to improve the performance of sparse matrix-vector multiplication and sparse matrix-matrix multiplication on a CPU-FPGA system.

2. "An Accelerator for Sparse Convolutional Neural Networks Leveraging Systolic General Matrix-Matrix Multiplication," [117] and its corresponding technical report [116] that presents our hardware accelerator for sparse convolutional neural network by proposing a novel hardware unit to perform the IM2COL transformation of the input feature map coupled with a systolic array-based general matrix-matrix multiplication (GEMM) unit.

3. “Near-Storage Processing for Solid State Drive Based Recommendation Inference with SmartSSDs.” [114], which builds an inference engine for deep learning-based recommendation systems using a SmartSSD device that features an FPGA and an SSD device. Our design offloads part of the computation to the FPGA on the SmartSSD, improving performance, and energy efficiency by doing the computation near the data.

1.3 Dissertation Organization

Chapter 2 provides background on sparse formats and sparse linear algebra kernels and then presents details on our synergistic hardware/software design for SpMV and SpGEMM kernels using a CPU-FPGA system. In Chapter 3, we first provide a primer on CNNs and sparsity-awareness designs. Then, we present our ASIC design for accelerating sparse CNNs inference task. Chapter 4 presents the end-to-end FPGA prototype of our accelerator for sparse CNNs. Chapter 5 evaluates the performance and energy efficiency of our ASIC and FPGA prototypes for various sparse CNNs. Chapter 6 concludes the dissertation and provide future directions.

CHAPTER 2
SPARSE LINEAR ALGEBRA ACCELERATION FOR MULTIPLE FORMATS USING A CPU-FPGA SYSTEM

There are a variety of applications from various domains that use linear algebra kernels, including multigrid methods [41], graph processing [65], and simulating physical body dynamics [58]. In many of these applications, the majority of the elements are zeros (more than 99%). Many compressed formats (i.e., sparse formats) have been explored to avoid storing the zero elements and performing computations on them. Most standard sparse formats involve indirection and irregular memory accesses to locate and match the non-zero values, resulting in low memory bandwidth for these sparse kernels. In response to this, hardware accelerators are utilizing specialized sparse formats that cater to specific operation and input sparsity patterns. By using custom formats, these hardware accelerators are able to access the non-zero elements with higher bandwidth and extract more parallelism. Despite its benefits, this method has two main disadvantages. First, sparse data are often subject to multiple operations, and each operation may require the data to be stored in a different format for optimal performance. Thus, the inputs must be preprocessed from other formats into a hardware-friendly format before computation can begin. Second, the inputs to the application may frequently change where each input has a different sparsity pattern. Hence, customizing the sparse data storage based on the sparsity pattern of the input requires an expensive preprocessing step for each input.

This chapter presents a software-hardware approach for high performance and adaptive computation of sparse matrix-vector multiplication (SpMV) and sparse general matrix-matrix multiplication (SpGEMM) kernels on a cooperative CPU-FPGA platform. In our design, which we call REAP, the CPU reorganizes the elements of a sparse matrix, allowing the FPGA to perform the computation with a high degree of parallelism. We pro-

pose a novel intermediate representation, which we call REAP intermediate representation (RIR), enabling the CPU to communicate the sparse data and the scheduling decisions to the FPGA. The preprocessed data in the RIR format, which is created by the CPU, removes all indirect accesses and index matching operations on the FPGA. Hence, the FPGA design can adapt to different sparsity patterns, dimensions, and data precision. To maximize performance, CPU tasks and FPGA tasks are overlapped. In contrast to prior research, our approach accelerates computation with multiple commonly used sparse formats without changing the FPGA's design, which is a significant improvement over approaches that require a custom design for each sparse format.

2.1 Overview of Our CPU-FPGA system for Accelerating Sparse Linear Algebra Kernels

Field Programmable Gate Arrays (FPGAs) promise to improve energy, throughput, and latency for sparse kernels, as they become more widely available in modern data centers. FPGAs are better than general-purpose architectures at exploiting fine-grained parallelism by allowing for application-specific custom memory accesses and compute engines. However, broadly realizing gains for sparse computation using FPGAs remains a challenge. Most standard sparse formats introduce indirect and irregular memory accesses, resulting in suboptimal memory throughput. The irregularity in the sparse operations also can introduce data dependencies that reduce the frequency and increase resource needs. These factors result in low compute utilization and limit the FPGA's ability to extract sufficient parallelism to achieve meaningful performance improvements over general-purpose architectures.

A large body of prior work has explored various methods to address these challenges. A common approach is to use a customized compressed format tailored to a specific FPGA's architecture. For example, recent work proposed a custom format for multi-die High Memory Bandwidth (HBM) FPGAs [35]. Another work proposed a scheduling technique that

partitions the sparse inputs into smaller batches that fit into the on-chip memory [119]. Similarly, SPAGHETTI [54] uses a pattern-aware software scheduler that analyzes the sparsity pattern and schedules the non-zero pairs of the inputs onto the fixed microarchitecture. While these studies successfully improved the performance of an FPGA for certain sparse kernels and architectures, their end-to-end performance degrades for standard sparse formats because of expensive preprocessing costs.

We propose a design with three key features to address some of the weaknesses of prior FPGA accelerators for sparse linear algebra kernels. First, our design supports a wide range of standard sparse formats. Instead of requiring a preprocessing step to convert the data from its original format to the format supported by the hardware, our design processes the input directly using the sparse format in which the data is stored. Second, our design is flexible to support both high precision inputs (*e.g.*, Float) as well as low precision inputs (*e.g.*, Int8) depending on the application's need. Relaxing the data precision of sparse kernels can result in significant performance gain [6]. Third, our approach is generic and thus can be applied to various sparse linear algebra kernels, not only a single sparse kernel.

Our design which we call it REAP is a cooperative CPU-FPGA system for sparse computation that combines the strengths of both the CPU and the FPGA. In REAP, the CPU reorganizes the sparse data into a format suitable for the FPGA to keep its computation units active. CPUs are good at manipulating small-scale, unpredictable memory access patterns as they have high clock rates and multiple levels of cache, while FPGAs with an application-specific routing and an abundant number of DSP units and on-chip memory are suitable targets to perform numerical computation on the reformatted streams of data.

The CPU identifies the matching elements that need to be multiplied and schedules the computation on the FPGA during the reorganization task. To communicate the reorganized data and scheduling information from the CPU to the FPGA, we developed a new intermediate representation called *REAP intermediate representation* (RIR). Our FPGA design consists of replicated pipelines with a large number of multipliers. The FPGA reads the

regular preprocessed data from the CPU in RIR format and performs the computation. We organize the entire computation in steps to facilitate parallel processing of the data reorganization by the CPU and the computation on the FPGA. When the CPU preprocesses the data for step k, the FPGA performs the computation corresponding to step $k-1$. This overlapped execution by the CPU and the FPGA further improves performance. The other important feature of our design is its ability to adapt to a new sparse format by simply modifying the software preprocessing task on the CPU. More importantly, it is not necessary to change the hardware on the FPGA to handle a new sparse format, which allows us to support widely used sparse formats with a single design. Besides, RIR can adapt to different sparsity patterns, input sizes, and data input precision.

2.1.1 Contributions

The contributions to this chapter can be summarized as follows:

- We propose a synergistic CPU-FPGA system for accelerating sparse linear algebra kernels whose inputs have a high percentage of zeros.
- We propose an intermediate data representation for communication between the CPU and FPGA that can be applied to multiple sparse kernels while supporting various sparse formats and different data precisions.
- We propose a high performance CPU-FPGA implementation for two important sparse problems namely, SpMV and SpGEMM kernels that supports multiple sparse formats and different data precisions.

2.2 Background on Sparse Formats and Sparse Linear Algebra Kernels

In this section, we first provide a brief primer on different sparse formats to store sparse matrices in a compressed form. We then provide background on sparse matrix-vector mul-

tiplication (SpMV) and sparse general matrix-matrix multiplication (SpGEMM). Finally, we review the high-level architecture of modern FPGA devices.

2.2.1 A Background on Sparse Formats

In a sparse matrix, the majority of the elements are zeros. The number of non-zero elements divided by the total number of elements in a matrix is termed the *density* of the matrix. It is inefficient to store a sparse matrix in the same format as a dense matrix. Thus, researchers have explored different compression schemes (sparse formats) to avoid storing zero elements [10, 11, 15, 26, 104]. Each sparse format is ideal under certain conditions, and no single format is superior for storing sparse matrices. The ideal format depends on different factors such as the sparsity pattern, the workload, and the hardware in use. Hence, it is desirable that accelerators support multiple formats. Otherwise, extra reformatting is required to put the data into the supported format.

In this dissertation, we discuss some of the most commonly used sparse formats, such as Coordinate format (COO), Compressed Sparse Row (CSR)[104], Diagonal (DIA), ELLPACK (ELL)[26], Bitmap Encoding [91], and Run-Length Encoding (RLC) [72]. Next, we explain each of these sparse formats.

Coordinate (COO). Coordinate format, commonly known as *COO* format [10] is the most primitive way to store the non-zero values of a sparse matrix. The COO format keeps a list of values and the coordinates (*i.e.*, row and column indices) of the non-zero elements (Figure 2.1(b)). The COO format has a minimal preprocessing cost since it only requires appending the non-zeros with their coordinates.

Compressed sparse row (CSR). The COO format redundantly stores row coordinates for every non-zero value. One way to remove these redundant row coordinates is by using an auxiliary array that keeps track of which non-zeros belong to each row. The values and columns of the non-zero elements are stored row by row in the two separate arrays. Then we store the starting offsets for all the rows in another array, commonly named *row_pointer*

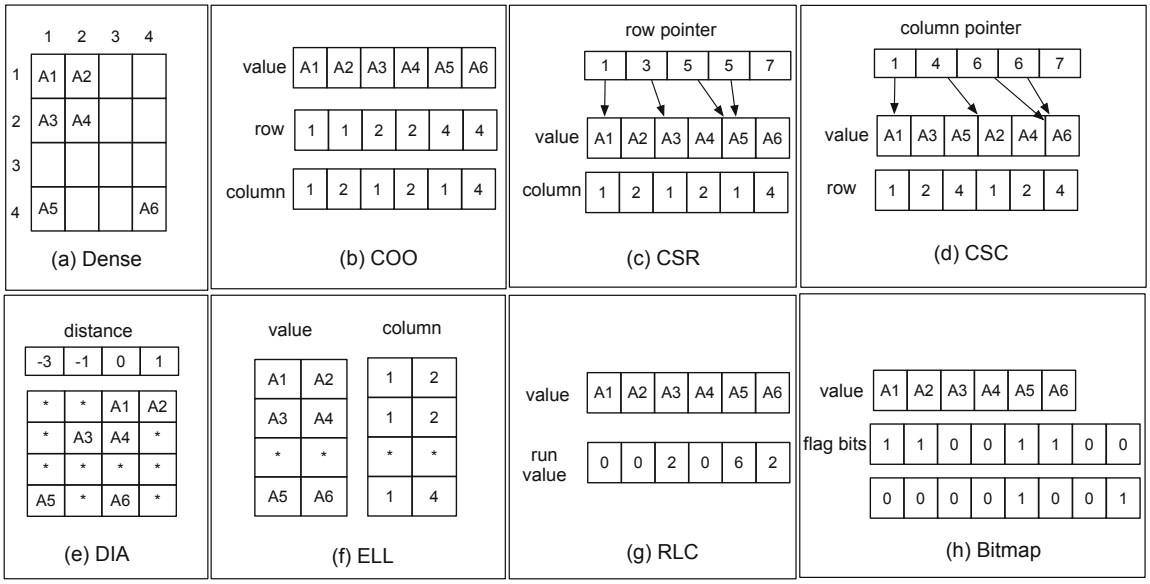


Figure 2.1: Different sparse format representations. (a) A dense representation of the example matrix (as a 2-dimensional array). (b) COO representation. (c) CSR representation. (d) CSC representation. (e) DIA representation. (f) ELL representation. (g) RLC representation. (h) Bitmap representaion. Here, * in the sparse representations indicates 0.

array. This method of storing the non-zero values is called Compressed Sparse Row (CSR) (*i.e.*, CSR in Figure 2.1(c)). The dual of the CSR format is called Compressed Sparse Column (CSC) that stores the column pointers instead of the rows, and the non-zero values are stored in column-major fashion in contrast to the row-major fashion in the CSR format (Figure 2.1(d)).

Diagonal (DIA). is a memory efficient format for sparse matrices where most of the non-zero elements are near the diagonal. The DIA storage format is specified by two arrays: *distance* and *value*. Here, *distance* is an integer array whose dimension is equal to the number of non-empty diagonals. Element *i* of the distance array represents the distance between the i^{th} diagonal and the main diagonal. A positive distance suggests the element is above the main diagonal, while a negative distance indicates the element is below the main diagonal. The main diagonal has a distance equal to zero. The value is a matrix with the number of rows equal to the number of rows in the dense representation and the number of columns equal to the number of non-zero diagonals. The i^{th}-column of the value matrix

stores the elements belonging to the i^{th} non-zero diagonal. They are stored in the rows corresponding to the dense representation. For example, the value of 1 belongs to the 1st diagonal, which has a distance of -3 from the main diagonal (see DIA in Figure 2.1(e)).

ELLPACK (ELL). is useful for matrices that contain a bounded number of non-zeros per row. ELL storage is specified by two arrays: *column* and *value*, which both have *N* by *M* dimension, where *N* is the number of rows of the matrix and *M* is the maximum number of non-zeros per row. The value and column indices of each non-zero element in row *i* is stored in the i^{th} row of the value and column arrays, respectively (see ELL in Figure 2.1(f)).

Run-length Encoding (RLC). Run-length encoding (RLC) [72] compressed a given sequence of values by replacing the continuing series of the same values with a single value that shows the number of repetitions (run). For a sparse matrix, the *run* indicates the total number of zeros before (or after) a non-zero value. Thus, in RLC, a sparse matrix is identified with a list of non-zero values and their run values (Figure 2.1(g)). Each run value indicates the total number of leading zeros before the next non-zero in the list. For example in Figure 2.1(g) there are two zero values between $A3$ and the previous non-zero element ($A2$) in a row-major order, therefore the run value for $A3$ is 2 (the third index).

Bitmap Encoding. Similar to COO, the Bitmap encoding [91] stores only the non-zeros elements in the matrix. Along with the non-zero values, we need one *flag bit* for *all* the elements in the matrix including both zero and non-zero elements (Figure 2.1(h)). The flag bit indicates whether an element is zero (*i.e.*, flag=0) or a non-zero (*i.e.*, flag=1). The bitmap encoding is effective for low to moderate sparsity percentages such as sparse neural networks.

2.2.2 A Background on SpMV and SpGEMM kernels

SpMV. In SpMV, we multiply a sparse matrix with a dense vector that produces an output vector that is also dense. Each output value is the result of a dot product between a row of the matrix and the input vector. A naive dense implementation of the SpMV ker-

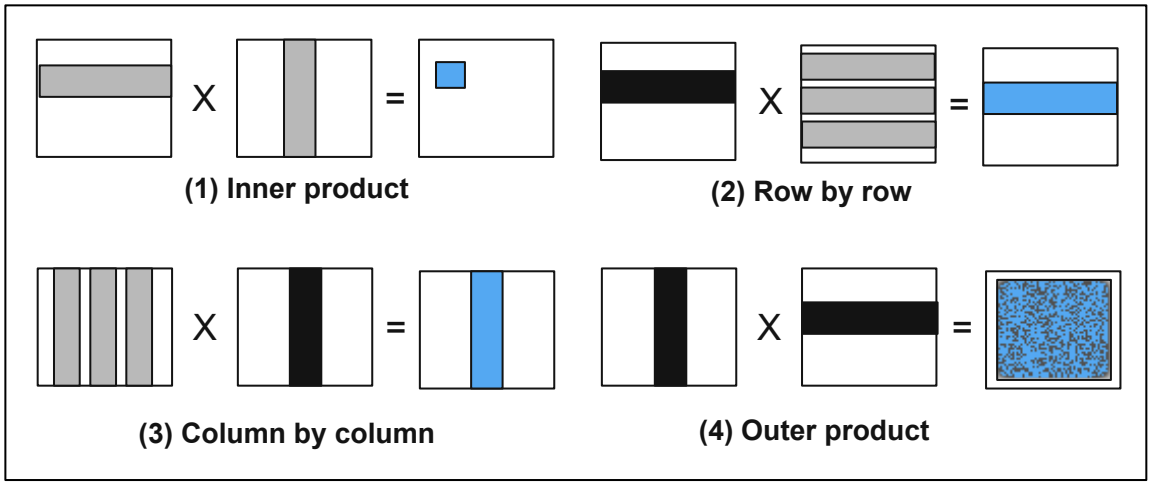


Figure 2.2: Four different formulations to perform general matrix-matrix multiplication. (1) Inner product. (2) The row by row method. (3) The column by column method. (4) Outer product. The stationary input for each formulation is shown in dark color.

nel performs computations on every element of a two-dimensional matrix, including the zero elements. The naive approach introduces high computational and storage overheads. Instead, the SpMV implementation computes the dot product using the sparse representation of the matrix (*e.g.*, CSR format). The algorithm avoids unnecessary computations of zero elements by iterating only over non-zero elements. However, this approach introduces pointer-chasing operations to access the elements in the vector. The column indices of the element in the sparse matrix are used as an index to load the appropriate element of the vector (line 7 in Algorithm 1).

SpGEMM. In SpGEMM, we multiply two sparse matrices, A and B, which results in another sparse matrix (C). Based on the order (*i.e.*, row-major or column-major) in which each input matrix is accessed, there are four different formulations for a SpGEMM, as shown in Figure 2.2. We compare each formulation from four different perspectives: (1) preprocessing requirements, (2) on-chip memory requirement for the inputs and the output matrices, (3) opportunities for parallelization, and (4) data reuse. A formulation has higher data reuse if it performs more multiply-accumulate (MAC) operations per memory read/write.

Row-by-column formulation. This formulation is also known as the *inner product*. In this formulation, each element of the output is calculated by performing a dot product between a sparse row from matrix A and a sparse column of matrix B (see the inner product in Figure 2.2). The two input matrices for the inner product are accessed in two different orders (*i.e.*, one matrix in row-major order and the other in column-major order). It needs a preprocessing step if the input sparse formats are not in the expected order. The inner product has low on-chip memory demands as it needs one row of the first matrix and one column of the second matrix to compute one element of the output matrix. The dot product computation for any two distinct elements of the output matrix can be performed in parallel. For the dot product, the indices are matched for the two input sparse vectors (sparse row of matrix A and sparse column of matrix B). Since the two input matrices are highly sparse, it is possible that none of the indices match, which results in data being read without any computation. Overall, the inner product has low data reuse since the MAC operations are performed only when the indices of the non-zero elements are matched.

Column-by-row formulation. This formulation is also known as the *outer product*. In contrast to the inner product, the outer product takes each column of A and a row of B and computes partial products belonging to the entire output matrix. Given a column vector of A and a row vector of B, the outer product multiplies a given element of the column vector with all elements of the row vector to produce partial products that correspond to a row of the output and accumulates them (see the outer product in Figure 2.2). Similar to the inner product method, the two input matrices are accessed in two different opposite orders. Unlike the inner product, the first matrix is accessed in column-major order, while the second matrix is accessed in row-major order. Hence, it needs a preprocessing step if the input matrices are not in the appropriate order. As this method produces partial products for the entire output matrix that need to be accumulated in each step, the on-chip memory requirement for an outer product is high. The partial products can be computed in parallel, but accumulation needs synchronization. All read items are used with multiplication. Hence,

this formulation has the highest data reuse.

Row-by-row formulation. In this formulation, the entire B matrix is read for each row of A to produce a single row of the output matrix. Each row of A is multiplied with every row of B, one at a time, and the partial products are accumulated (see row by row in Figure 2.2). This formulation is often known as Gustavson's algorithm [46]. The core computation of a row-by-row formulation is a *scalar-vector product*. Unlike inner-product and outer product, the row-by-row formulation access both matrices in row-major order. Therefore, it can process two inputs in the same format. In contrast to the outer product, a single output row is computed at a time. Hence, the memory requirement is not as high as the outer product. However, it does require one matrix to be read multiple times. Further, all the rows of the sparse output matrix can be computed independently and in parallel.

Column-by-column formulation. In this formulation, the matrix A is read multiple times for each column of B to produce one column of the output matrix. For a given column of B, we multiply each column of A with that column of B to compute partial products and accumulate them to produce the column of the output matrix (see column by column in Figure 2.2). Both matrices are accessed in the column-major order. The memory requirements, parallelism, and data reuse is similar to the row-by-row formulation.

2.3 Synergistic CPU-FPGA Acceleration

In this section, we describe the general architecture of REAP for accelerating sparse linear algebra kernels. We then describe the instantiations of REAP for two kernels: SpMV and SpGEMM. The unique aspect of our approach to accelerating sparse computation with REAP is that we divide the computation of sparse linear algebra kernels into two main tasks. The first task involves discovering the position of non-zero elements in the sparse inputs and reformatting them as regularized data in the intermediate data representation. The second task performs numerical computation on the reformatted data in the intermediate data representation. REAP uses the CPU for the first task and the FPGA for the second.

Further, the computation on the CPU and the FPGA are overlapped. When the CPU pre-processes the data for step k, the FPGA performs the computation corresponding to step $k-1$. Thus, the CPU and FPGA can execute in parallel, which improves performance.

For the CPU to communicate the reformatted data to the FPGA, we developed the **REAP intermediate data representation (RIR)**. Our CPU-FPGA synergistic approach using the RIR format has three main advantages. First, reformatting the sparse data by the CPU removes all indirect accesses and index matching operations. It also allows large bursty data transfers from the CPU to the FPGA. A second benefit of the RIR format is that the same custom hardware can support multiple sparse formats. The FPGA computes using inputs in RIR format. Thus, multiple sparse formats can be supported with the same FPGA design by only changing the software. We can avoid long re-compilation times and performance regressions when adjusting the FPGA design to support a new sparse format. Additionally, it allows our approach to flexibly support different sparsity patterns, input sizes, and data input precision.

RIR has two important features - it is extensible and supports information about both data and metadata. We designed RIR to be extensible. RIR can be used to express information in multiple sparse kernels. For a given sparse kernel, the RIR format is designed with respect to the core operation of that kernel. While the core features of the RIR format remain the same for different sparse kernels, each sparse kernel has its own unique RIR structure. For the FPGA part of the design, this feature is essential to achieving optimal performance. We describe the RIR format for SpMV and SpGEMM in Sections 2.3.1 and 2.3.2.

The information exchanged between the CPU and FPGA is aggregated into a collection called the RIR bundle. In this way, we can amortize the cost of communication. RIR supports both data and metadata bundles. The data bundle includes the values and coordinates of the elements required for the kernel's core computation. The metadata provides extra information that allows the FPGA to manage the partial products and schedule the

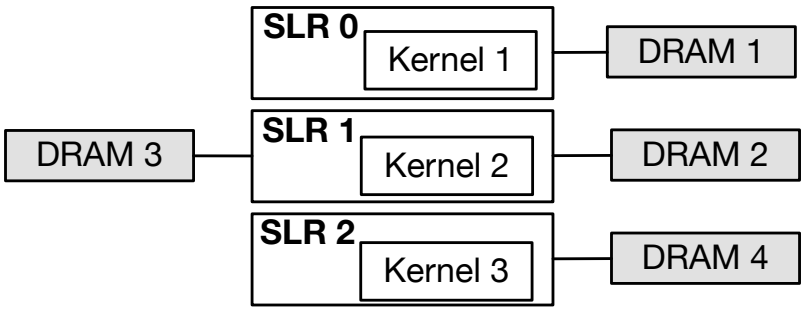


Figure 2.3: The floorplan of XCU200 FPGA.

work among different processing elements. This requires co-designing the software and hardware such that the software on the CPU is aware of the FPGA design.

Computation on the FPGA in REAP. Modern FPGAs use Stacked Silicon Interconnect (SSI) technology to build large FPGAs with an affordable power envelope. Figure 2.3 shows the floorplan for the Xilinx XCU200 FPGA device features on the Alveo U200 board. XCU200 FPGA comprises three Super Logic Regions (SLRs) that are stacked on top of each other using SSI technology. Any crossing (connection) between the SLR regions can negatively impact the operating frequency. Our FPGA design consists of replicated pipelines, with each pipeline consisting of multiple processing elements (PEs). To achieve high frequency and memory bandwidth, each pipeline is assigned to a different SLR and memory bank (see Figure 2.3). The FPGA reads the inputs that are preprocessed by the CPU and stored in RIR format. Figure 2.6 and Figure 2.8 present our FPGA designs for SpMV and SpGEMM, respectively.

There are two sources of parallelism in the FPGA design: the parallel pipelines and the independent PEs inside each pipeline. Keeping both the pipelines and the PEs busy is the key to performance for the FPGA part of the design. The only way to accomplish this is to access the data with high memory bandwidth. The three essential requirements for achieving peak memory bandwidth for FPGAs are: large input/output data width, sequential and predictable memory accesses, and reading the data in large burst sizes [84]. Preprocessing of sparse data by the CPU eliminates indirect accesses and allows the FPGA to achieve

near-peak memory bandwidth.

Preprocessing performed by the CPU in REAP. In REAP, the CPU processes the sparse inputs in their native compressed format and transforms them into a more *hardware-friendly* RIR format. The two main tasks done by the CPU involve identifying the non-zero elements from the compressed sparse data for a particular sparse kernel and computing the metadata that allows the FPGA to complete its numerical computation with high performance.

Identifying non-zero elements and index matching often introduces indirect memory accesses. We found that CPUs are better at manipulating small-scale, unpredictable memory access patterns with their multiple levels of caches. Besides, in REAP, the CPU does not perform any arithmetic operations. Instead, numerical computations are performed by the FPGA. In most modern CPUs, the same memory bandwidth can be achieved by operating at half the base frequency [107]. Thus, our design allows the CPU to operate at a lower frequency and still maintain the memory bandwidth, which helps reduce the energy consumption on the CPU side. The flexibility of REAP in supporting various sparse formats comes from the software on the CPU. Each sparse format needs a unique decoding step; all is done in software.

When building RIR bundles in software, there are two levels of parallelism to target: pipeline parallelism and PE-level parallelism within the pipeline on the FPGA. Additionally, CPU preprocessing is done in parallel using multiple threads. The task of building RIR bundles for different pipelines can be performed independently and in parallel on multiple threads since they are independent. Building RIR bundles for PEs within a pipeline can also often be executed in parallel. CPU preprocessing is specific to the kernel being accelerated. We describe the regularization of data with preprocessing performed by the CPU for SpMV in Section 2.3.1 and for SpGEMM in Section 2.3.2.

Overlapping the work done by the CPU and the FPGA. To facilitate the parallel execution of data reorganization by the CPU and computation on the FPGA, we organize

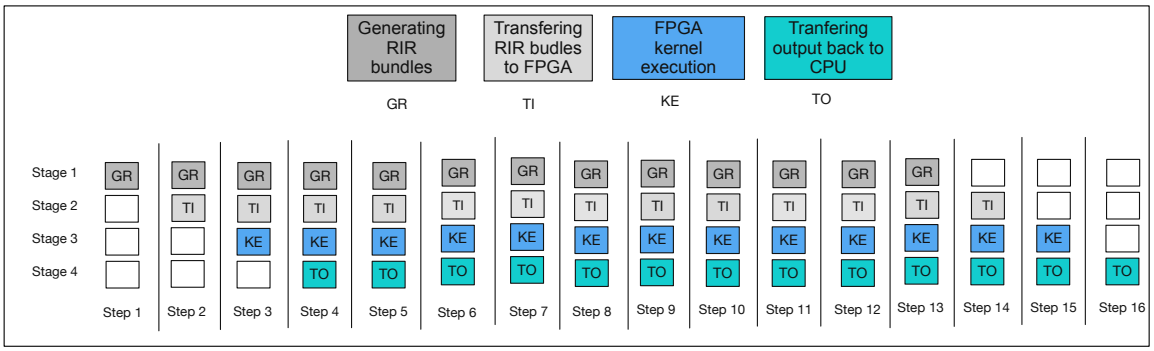


Figure 2.4: Overlapping of the tasks performed by the CPU and FPGA tasks in REAP. The pipeline consists of four stages. (1) Generation of RIR bundles (GR). (2) Transfer of RIR bundles to the FPGA (TI). (3) FPGA kernel execution (KE). (4) Transfer of the output back to the CPU (TO). All four tasks can be performed in parallel in the steady state (*i.e.*, steps 4-13).

the entire computation in steps. When the CPU preprocesses the data for step k, the FPGA performs the computation corresponding to step $k-1$. Figure 2.4 shows different steps performed by the CPU and the FPGA. There are four tasks: generating the bundles (done by the CPU), transferring the RIR bundles to the FPGA memory, performing the computation (done by the FPGA), and transferring the result back from the FPGA memory to the CPU. After the initial steps, all four tasks can be performed simultaneously. The tools for programming modern FPGAs provide asynchronous application programming interfaces (APIs) for data transfers and kernel invocation. In addition, we use double buffering to facilitate parallel execution by the CPU and the FPGA. The CPU uses one set of buffers and the FPGA uses the other.

Another design consideration in overlapping the computation on the CPU and the FPGA concerns the choice of input sizes for each computation step. A large chunk size reduces the total number of steps and reduces the overlapping opportunities, especially for smaller input matrices. In contrast, larger input sizes are preferable for achieving higher memory bandwidth [84]. We choose the input size based on the profiling information for our prototype.

2.3.1 Instantiation of REAP to Accelerate SpMV

In a sparse matrix-vector multiplication (SpMV) kernel, we multiply a sparse matrix with a dense vector to produce a dense vector. Algorithm 1 presents the SpMV formulation that we use. Each output value is computed with a dot-product between a row of the sparse input matrix and the input vector (lines 10-13 in Algorithm 1). Due to sparsity, the size of the vectors used for the dot-product can vary with each row. In our design, the CPU reads the sparse input matrix and organizes the vectors for a dot-product. The FPGA computes the dot product using the preprocessed vectors. The CPU and the FPGA can independently process different rows of the sparse input matrix, allowing their computation to be overlapped.

The RIR format for SpMV. The key operation in SpMV is the dot-product of a row of the sparse matrix ($ValA_r$ in Algorithm 1) and the input vector. The result of each dot-product is one element of the final vector. The number of multiply-add operations needed for computing one element of the result vector is proportional to the number of non-zeros in a row of the sparse matrix. The RIR format for SpMV is designed based on the dot-product computed by the SpMV kernel.

We design both data and metadata bundles tailored for SpMV in the RIR format. Figure 2.5(b) and Figure 2.5(c) shows the RIR format for metadata and data bundles. Each data bundle consists of N pairs of non-zero values from the sparse matrix and the input vector. The value of N is a design parameter. For each data bundle, there is a corresponding metadata bundle. The metadata bundle contains two pieces of information. First, it carries one bit of information (*End* in the Figure 2.5(b)) that indicates if this bundle is the last bundle belonging to the same output element or not. This becomes important when the number of non-zeros in a row is greater than N. In those cases, we divide the non-zeros into multiple bundles. The last bundle in a series is marked with a 1 in the *End* section, while the other bundles are marked with a 0. Besides the *End* bit, each bundle also includes scheduling information (*PE ID*). This enables the FPGA to route the data bundles to the

different processing elements (PEs). All bundles that contribute to the same output element are processed by the same PE, which eliminates the need for any communication between PEs. The CPU is aware of the number of PEs in each pipeline to do the scheduling task properly.

An invariant of the RIR format for SpMV is that every row in the input sparse matrix has at least one data bundle. We maintain this invariant to avoid storing the row index in the metadata. When there is a row with all zeros in the input sparse matrix, there will be a data bundle filled with a pair of zeros (see bundles 3 in Figure 2.5(c)). We found that maintaining the row index in the metadata degrades performance significantly compared to our design with a single data bundle with a pair of zeros.

Optimal N depends on the sparsity pattern of the sparse matrix. Large values of N can reduce the metadata overhead. However, for rows with less than N non-zeros per row, the remaining pairs are filled with zeros (For example, see bundles 2 and 6 in Figure 2.5(c)). We set N as 4 in our prototype based on empirical analysis. The size of the data bundle is dependent on the precision of the input data element. The size of the bundle is $N * 2 * 4$ bytes if the input is single-precision float and there are N pairs of non-zero values. When $N = 4$ with a single-precision float as the data type, the size of each data bundle is 32 bytes.

Packing RIR bundles to build wide vectors. Using wide AXI ports for accessing external memory improves FPGA memory bandwidth. A port width of 512 bits was discovered to be optimal for memory performance. By packing multiple RIR bundles together, we can build one large input vector. For single-precision inputs with 32-byte data bundles, we can pack two data bundles into a 512-bit vector. For a 16-bit integer and an 8-bit integer, we can pack 4 and 8 data bundles per vector, respectively.

The FPGA design. Figure 2.6 shows the details of the SpMV FPGA architecture. Our design consists of three pipelines (Figure 2.6(a) shows only one of the three pipelines). Each pipeline consists of five stages (fetch, scheduler, dot-product, merge, and write) that

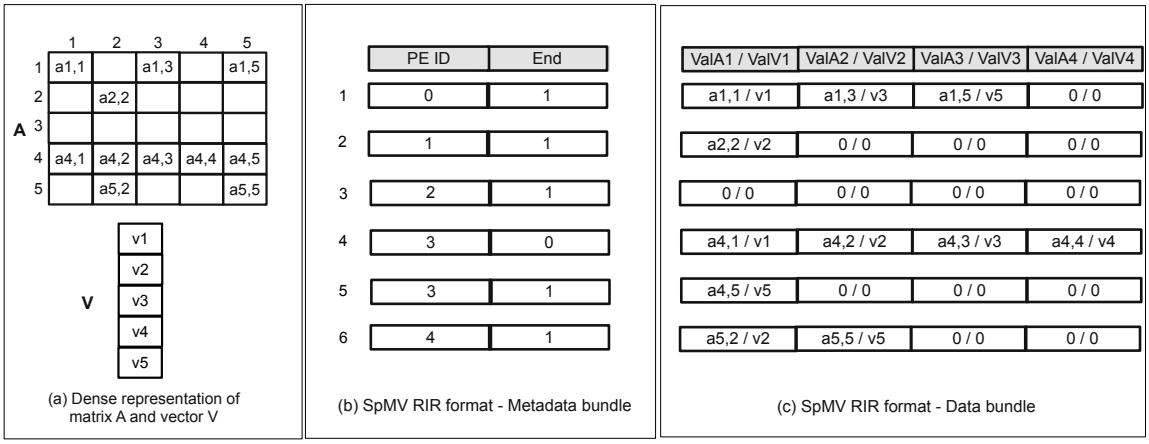


Figure 2.5: (a) The input matrix A and the dense vector V are both in their dense representation. (b) The RIR metadata bundle generated for the SpMV operation on sparse matrix A and dense vector V in (a). (c) The RIR data bundle for this example.

process the input data in the RIR bundle. Each fetch unit reads the RIR bundles from a bank of DDR4 memory. It is connected to one advanced extensible interface (AXI) channel to read data from memory in burst mode, which is feasible because of the preprocessing done by the CPU. Further, REAP's design uses a wide data bus of 64 bytes. Using a wide data bus helps the FPGA reach its peak memory bandwidth. The fetch unit passes the data to the scheduler, which distributes the RIR bundles among the PEs by examining the metadata embedded in the bundle by the CPU. Each bundle is assigned to one of the PEs. In REAP's SpMV design, each pipeline has up to 16 PEs. Each PE then performs the dot product. Each PE has 4 multiplication units with an add tree (see Figure 2.6(b)). The result of multiplication from each bundle is sent to the merge unit to be added to the result of the previous bundles. Once the merge unit receives the last bundle of a row vector, it sends its result to the write unit. The *write* unit reads the result received from different PEs connected to it and packs the result into a wide vector, and then writes the results to memory. All stages are connected by FIFO streams, which enables them to operate concurrently.

Preprocessing on the CPU. CPUs preprocess sparse input matrices and dense vectors to generate RIR bundles, which keeps FPGA pipelines busy. The CPU in the preprocessing

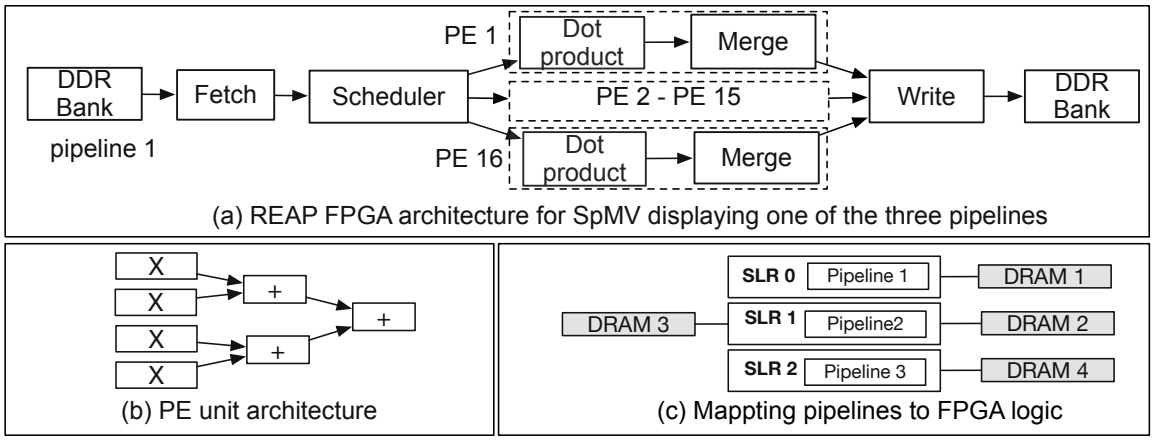


Figure 2.6: (a) Our FPGA design for SpMV. (b) The PE architecture for performing the dot product. Each PE has 4 multiply units with an add tree. (c) Illustration of how each pipeline maps to a different super logic region (SLR) on a modern FPGA. Each pipeline accesses a different DRAM memory bank.

step needs to match the non-zero elements of the sparse matrix with the elements in the input vector. Each non-zero value in matrix A with a column index of c is matched with the element in row c of the input vector. To compute one element in row R of the output vector, we need B data bundles where B can be calculated as $B = \lceil \frac{NNZ_R}{N} \rceil$, where N is the number of non-zero pairs in each data bundle and NNZ_R is the number of non-zeros in row R.

Our design schedules the bundles to PEs in a round-robin fashion. While using a dynamic scheduling algorithm can improve the load balance between the PEs in some cases, it can change the order in which the PEs compute the output elements. Thus, to combine the output elements and write them to the memory, we should include the output index (*e.g.*, the row index) in the metadata bundle, which can introduce extra overhead. In addition, a more complex scheduling algorithm can also increase the CPU execution time.

The CPU preprocessing itself to generate RIR bundles happens in parallel using multiple threads. Each CPU thread processes an equal number of rows from the input matrix. To produce a single array of RIR bundles for the FPGA, the outputs of the different threads must be merged. Each CPU thread writes to the different parts of the single output array.

Algorithm 1 SpMV formulation.

1: **procedure** SPMV($Input(SpA, VecV), Output(VecC)$)
2: **for** each row r in SpA **do**
3: $NNZ[r] \leftarrow GetNNZ(SpA, r)$
4: $ColA_r \leftarrow GetColIndices(SpA, r)$
5: $ValA_r \leftarrow GetValues(SpA, r)$
6: **for** j = 0 to $NNZ[r]$ **do**
7: $ValV_r[j] \leftarrow VecV[ColA_r[j]]$
8: **end for**
9: $Sum \leftarrow 0$
10: **for** i = 0 to $NNZ[r]$ **do**
11: $Sum \leftarrow Sum + (ValA_r[i] * ValV_r[i])$
12: **end for**
13: $VecC[r] \leftarrow Sum$
14: **end for**
15: **end procedure**

We calculate the total number of bundles for each thread and pass them as the starting index for each thread. The starting indices can be easily calculated for most sparse formats. For example in CSR, the $row\_pointer[i] - row\_pointer[j]$ gives the total number of non-zero elements between row index j and row index i. DIA and ELL formats have a fixed number of non-zero elements in each row.

We support any input size for the input vector and sparse matrix as opposed to prior work that maintains the entire dense vector on FPGA on-chip memory [66, 147].

2.3.2 Instantiation of REAP to Accelerate SPGEMM

From the perspective of acceleration, SpGEMM is more complicated than SpMV as it involves two sparse input matrices and a sparse output matrix. Unlike SpMV, the number of non-zeros in the output is input dependent in SpGEMM. A SpGEMM kernel consists of two main tasks: multiply and merge (also known as accumulation). The *multiply* task generates partial products. A partial product is the result of multiplying a non-zero value of input A with a corresponding non-zero value of input B. A match occurs when a *column*

Algorithm 2 SpGEMM formulation.

```
1:  procedure SpGEMM(Input(SpA, SpB), Output(SpC))
2:      for each row r in SpA do
3:          NNZ[r] ← GetNNZ(SpA, r)
4:          ColA_r ← GetColIndices(SpA, r)
5:          ValA_r ← GetValues(SpA, r)
6:          for j = 0 to NNZ[r] do
7:              ValA[j] ← ValA_r[j]
8:              NNZB[j] ← GetNNZ(SpB, ColA_r[j])
9:              ColB_j ← GetColIndices(SpB, ColA_r[j])
10:             ValB_j ← GetValues(SpB, ColA_r[j])
11:             RowC_r ← GetRow(SpC, r)
12:             for i = 0 to NNZB[j] do
13:                 PPVal ← ValA[j] * ValB_j[i]
14:                 PPCol ← ColB_j[i]
15:                 RowC_r ← Merge(RowC_r, PPCol, PPVal)
16:             end for
17:         end for
18:     end for
19: end procedure
```

index of A matches a *row* index of B. The *merge* task sums up all partial products that have the same coordinates (*i.e.*, row and column) and produces the final result.

We use the row-by-row formulation (also known as Gustavson algorithm [46]) in our design because the partial products belong to a row of the output rather than the entire matrix (as with the outer product [97]). Section 2.2.2 reviews four formulations for multiplying two matrices and their advantages/disadvantages. Although explored by prior work [97], the outer product formulation offers good throughput for the multiply task but has poor throughput for the merge task. A high overall throughput requires balancing the multiply and merge tasks. Our design with the row-by-row formulation has less complexity in maintaining partial products and performing the merge task and it better balances the merge and multiply tasks than outer product formulation.

Algorithm 2 illustrates a row-by-row formulation for multiplying two sparse matrices A and B. Intuitively, each row of A is compared with all the rows of B, matching elements

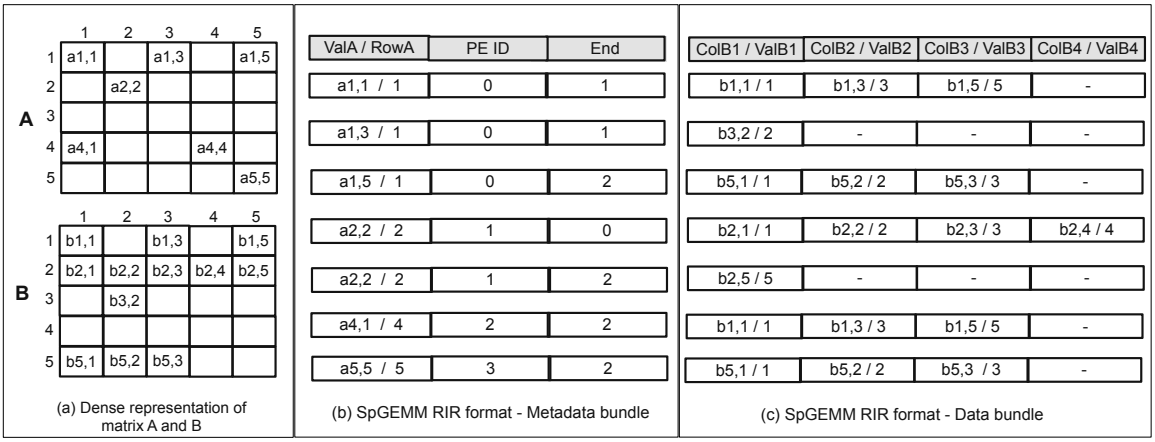


Figure 2.7: (a) Two input matrices A and B in their dense representation. (b-c) The RIR metadata and data bundles for this example. $a1,1$ matches with the first row of matrix B. The three values in the first row of B are $b1,1$, $b1,3$, and $b1,5$. Since each bundle holds four pairs, one pair is unused.

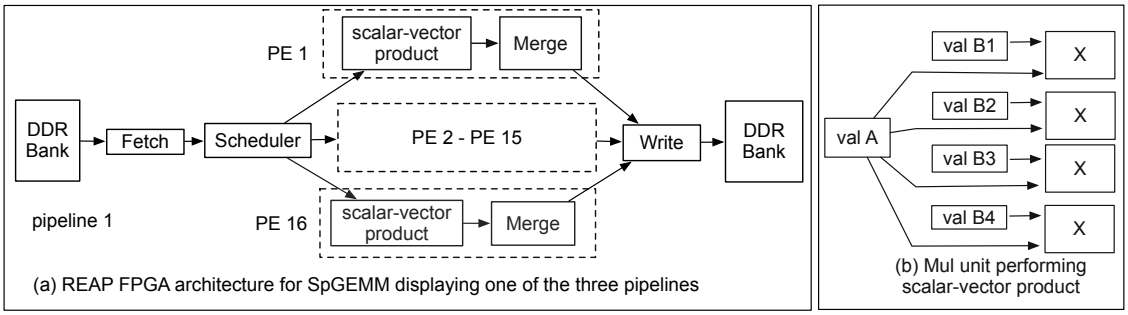


Figure 2.8: (a) FPGA design for SpGEMM. (b) The architecture of the multiply unit performing dot product.

(*i.e.*, when the column index of an element in A and the row index of an element in B are equal) are multiplied to generate partial products, and the partial products belonging to the same column index are accumulated to produce a row of the final result matrix. The A-matrix is read once, and the B-matrix is streamed into the FPGA for each row of A. Given that A and B matrices are sparse, it is not necessary to stream all the rows of B for a given row of A. When we read a row of A, we identify the column indices of the non-zero elements in that row and only stream those rows of B that match one of the column indices of A (lines 3-5 in Algorithm 2). For example, if there is only one non-zero element in a row

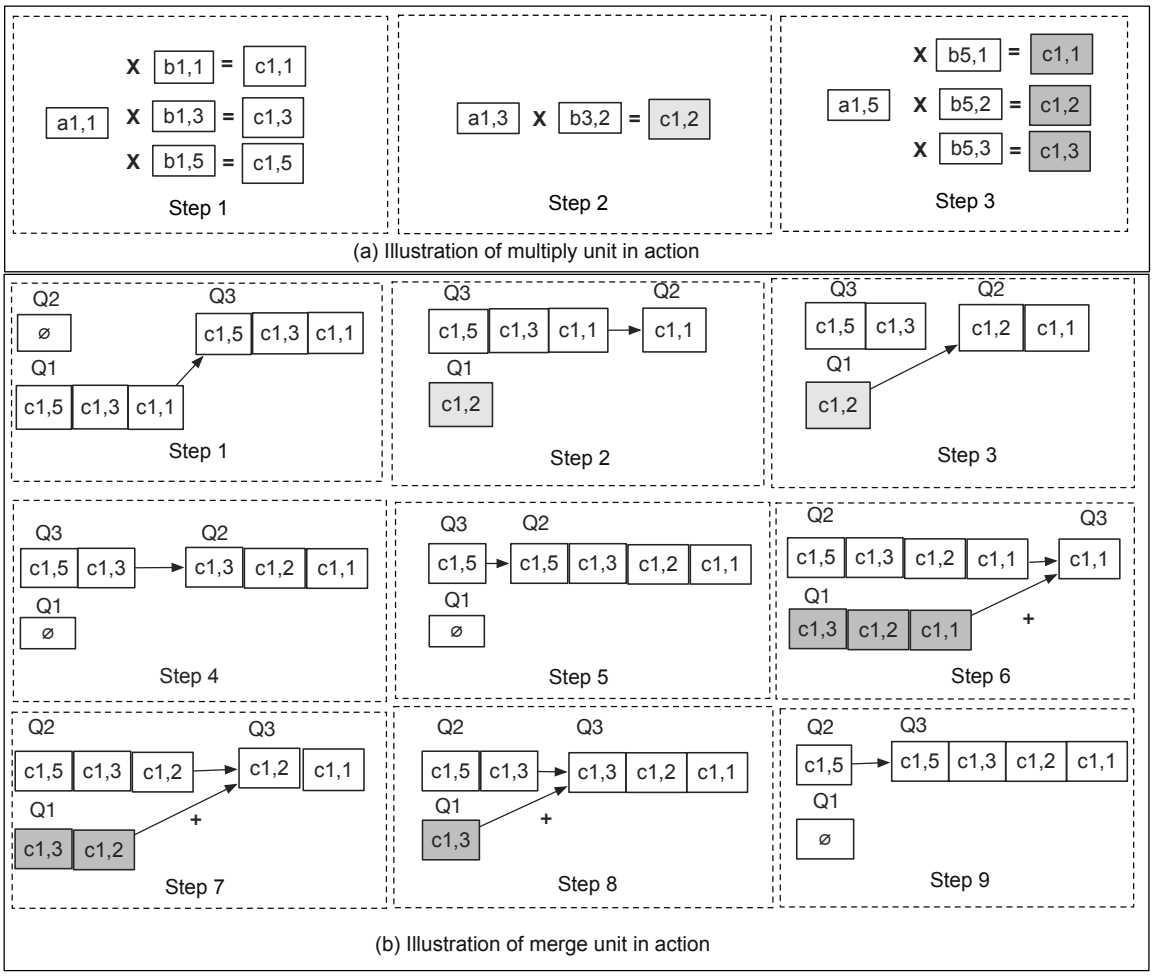


Figure 2.9: (a) The multiply unit in action in various steps while processing the first row of the result matrix from Figure 2.7(a). This involves the first three bundles from the example in Figure 2.7(b-c). (b) The merge unit in action in various steps while processing the first row of the result matrix from the example in Figure 2.7.

of A, it is unnecessary to stream all rows of B. Just streaming one row of B that matches the column index of the single non-zero element of A is sufficient.

The main tasks in SpGEMM are: (1) extract the matched non-zero elements from the two sparse matrices, (2) multiply the matched elements (*i.e.*, compute partial products), and (3) merge the partial products to produce the final result. Our design uses the CPU for the first task, and the FPGA for the rest. Similar to SpMV, the CPU reorganizes the input data as RIR bundles to make it easier for the FPGA to attain higher throughput computation.

The RIR bundles for SpGEMM. Like the SpMV design, we have data and metadata

RIR bundles. For each data bundle, there is a corresponding metadata bundle. The core computation of our SpGEMM formulation is a scalar-vector product. In our design, the data bundles contain the vector of non-zero elements, and the metadata bundles contain the scalar element. It is essential to include both value and column indices in each bundle for SpGEMM. To compute the final result, the column indices are required to merge the partial products.

Figure 2.7(a) shows an example of two matrices in their dense formats for illustration. In our design, the inputs are stored in their native compressed format (*e.g.*, CSR). The CPU preprocesses the data and creates RIR bundles shown in Figure 2.7(b-c). Each RIR metadata bundle includes (1) the row identifier and the value of a non-zero element of the matrix A, (2) the identifier for the PE that processes the bundle, (3) two bits (*End* bits) to encode three values representing whether we have seen all rows of B for a given row of A or not. A 0 for the *End* bits encodes that there are more bundles with a given row of B for a non-zero element of A. A value 1 encodes the end of bundles with a given row of B for a non-zero element of A. A value 2 encodes the end of bundles corresponding to all rows of B for the whole row of A. As the values of matrix B are streamed for every row of A, this metadata enables our design to handle matrices of any size. The data bundles include up to N non-zero elements of B that should be multiplied with the element from matrix A. For each non-zero element, we store both value and its column index.

Similar to our SpMV design, the size of the RIR bundle depends on the data types used for the column indices and the value. In our prototype, we use 32-bits for the column indices, which allow us to support matrices with dimensions as large as 2^{32}. The number of elements in each data bundle (N) is a design parameter, which is chosen based on the input data precision and the sparsity pattern of the input matrices. Based on profiling information, $N = 4$ is used in our design for single-precision inputs.

Our FPGA design for SpGEMM. Figure 2.8(a) provides the FPGA architecture for SpGEMM. It consists of three replicated pipelines. Each pipeline has five units that process

the data in RIR bundles (*i.e.*, fetch, scheduler, multiply, merge, and write). All units are connected with FIFOs, which allow concurrent execution. The *fetch* unit reads the bundles and passes them to the scheduler. The *scheduler* sends the bundles to their assigned PE using the scheduling information included in the RIR bundle. Each pipeline includes 16 *PEs* that perform *multiply* and *merge* tasks.

Each pair of data and metadata bundle received by each PE includes all the elements for a *scalar-vector* product. This significantly simplifies the job of the FPGA by eliminating the need to match the indices, which otherwise would require a costly content-addressable memory (CAM) in the FPGA.

The multiply unit performs a scalar-vector multiplication to produce the partial products for the i^{th} row of matrix C. This is done by multiplying the value of A (included in the metadata bundle) with all the B's elements in the data bundle. Each multiply unit in a PE performs up to N multiplications in parallel. Effectively, 192 multiplications can happen in parallel per cycle in the entire design (3 pipelines * 16 PEs per pipeline * 4 multipliers per PE = 192). The partial products are then sent to the merge unit with their corresponding index (*e.g.*, B's column identifier). The merge unit accumulates all the incoming partial products with the same column index.

When the CPU creates the bundle for a non-zero element of A, it orders the corresponding non-zero elements of B based on the column indices. Hence, the partial products arriving at the merge unit corresponding to each non-zero element of A are sorted by their column indices. We need to accumulate partial products belonging to the same output element. We use three queues (Q1, Q2, and Q3) to perform this merge and accumulation. The queue Q1 contains the partial products produced by the multiply units. The previously accumulated products corresponding to a row of B are in Q2. In each cycle, the merge unit gets an element from Q1 and Q2 and merges the result into result queue Q3. If the column indices of the elements at the front of Q1 and Q2 match, the values from both queues are removed, their values are added, and their result is pushed to Q3. If the column indices of

the elements at the front of Q1 and Q2 do not match, the value with the smaller column index is popped and pushed to Q3. While Q1 always stores the incoming partial products, we alternate Q2 and Q3 as inputs and outputs depending on whether we have completed processing a row of B, which avoids unnecessary copies of queue elements.

Figure 2.8(b-c) illustrates the operation of the multiply and merge units for the example from Figure 2.8. Finally, the *write* unit writes a complete row of the output from each PE in the order they are received.

When multiplying two sparse matrices, the number of non-zeros in a row of the result matrix is not known a priori. We address this issue by maintaining fixed-size queues in the merge unit and recomputing the results on the CPU when the number of such partial results exceeds the queue size. To store partial results for each row, we use a 1024 entry queue on the FPGA. Similarly, the CPU also uses the same sized data structure to store the results for each row. When the number of non-zero partial results in a given row exceeds the size of the queue, the FPGA informs the CPU by setting a bit in the output data. Subsequently, the CPU discards the results computed FPGA for that particular row and recomputes the results on the host.

Preprocessing performed by the CPU. The two main tasks of the CPU are (1) to access the elements in their compressed form and (2) to match the elements from the two input matrices. Each data bundle includes the rows of B that match with a non-zero element of matrix A. To compute the i^{th} row of the result matrix (C), we need M *scalar-vector* multiplication where M is the number of non-zero elements in a i^{th} row of A. All bundles belonging to the i^{th} row of matrix A are scheduled to the same PE. Similar to SpMV, the bundles are distributed among the PEs in a round-robin fashion.

We use multiple threads to parallelize the generation of RIR bundles, like SpMV design. Each thread handles a fraction of the rows of the A matrix. To handle the uneven distribution of non-zero elements, we first generate the metadata bundles. Since there is a one-to-one mapping between the data and metadata bundles, we can use the information

Table 2.1: The CPU, and FPGA configurations.

Platform	Configuration
CPU Intel Xeon 6130	48 cores, 2.6 GHz (base), 192 GiB DDR4 Cache(KB) L1:32K, L2:1024K, L3:19712K
FPGA Xilinx Alveo-U200	1,182K logic elements, 2280 DSP blocks 25-Mbits BRAM, 64 GB DDR4, Max memory bandwidth 77 GB/s

Table 2.2: FPGA resource utilization for SpMV and SpGEMM designs.

Application	LUT	BRAM	URAM	FF	DSP
SpMV	11%	24%	0%	10%	16%
SpGEMM	35%	30%	3%	12%	24%

(the number of metadata bundles) when generating the data bundles. When generating the metadata bundles, we use separate arrays for each thread and then combine their results into one large array containing all the bundles' metadata. While this approach needs extra data copies, overall, its performance is better than using a single thread process for generating the metadata.

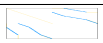
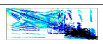
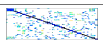
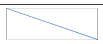
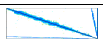
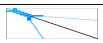
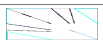
2.4 Experimental Methodology of REAP

Prototype. We have built an end-to-end prototype of REAP for SpMV and SpGEMM and evaluated it on the Xilinx Alveo U200 card. Alveo U200 features the XCU200 FPGA, which consists of three Super Logic Region (SLR). The SLR regions are combined to build a large device with an affordable power envelope using Stacked Silicon Interconnect (SSI) technology. Figure 2.6(c) shows the floorplan for the Xilinx XCU200 FPGA device. The three SLR regions in XCU200 FPGA are connected to different DRAM banks. Our design uses all three SLRs by mapping each of the three pipelines to a different SLR (Figure 2.6(c)). This allows our design to utilize multiple DDR banks and also attain high operational frequency by avoiding the crossing between different SLRs. Furthermore, we also use wide data buses and large burst transfers to attain the highest efficiency for the memory controller on the FPGA.

We developed the FPGA designs using the Xilinx HLS toolchain. We used appropriate pragmas in our programs specifying the desired microarchitecture. We implemented the streaming interface between different components of a pipeline using the Xilinx *HLS Stream* pragma. We experimented with *unroll* and *pipeline* factors for each individual loop to attain the best possible frequency. Table 5.2 shows the FPGA specification. The CPU and the FPGA communicate through PCIe. Our FPGA uses a Gen3 x16 PCIe interface. After place and route, FPGA designs for SpMV and SpGEMM can operate at 250 MHz and 200 MHz on the Alveo U200 card, respectively. Table 2.2 reports details on resource utilization on the FPGA for SpMV and SpGEMM designs for the single-precision inputs. Our designs created enough pipelines and processing elements to balance the CPU's and FPGA execution times.

CPU Baseline. We evaluate REAP design by comparing it to the well-optimized sparse kernels in Intel Math Library (MKL) [3]. Table 5.2 provides details on the CPU used for the comparison. We use the same CPU for the CPU execution of the REAP. To evaluate the SpGEMM kernel, we calculate $C = A^2$, which is a standard method for evaluating SpGEMM performance [53, 54, 120, 148]. Both SpMV and SpGEMM matrices use single-precision floating-point for the evaluation except when we explicitly specify the data width. To test the ability to support various sparse formats, we evaluate REAP for three sparse formats: CSR, DIA, and ELL. Intel MKL does not directly take these formats as inputs. Hence, our evaluation includes the time taken to convert the matrix in these formats into CSR. We also compare the performance of REAP for 16-bit and 8-bit integer input data. We mainly used twenty-one sparse matrices from the SuiteSparse matrix collection [27] listed in Table 2.3. Most of these matrices have been used by prior work in this domain [53, 54, 89, 97, 120]. These matrices vary in size, density, and sparsity patterns. We use wall clock time to measure the end-to-end performance, including the time for data reorganization on the CPU, data transfer, FPGA computation, and transferring the result from the FPGA to the CPU's memory.

Table 2.3: Matrices from SparseSuite [27] used in our evaluation. We use the ID to refer to specific matrices in the evaluation. [1] NNZ: the number of non-zero entries. [2] zeros are shown with black dots as opposed to the rest of the benchmarks.

Name	ID	Dimension	NNZ[1]	Kind	Sparsity pattern
cont-300	SA	180K X 180K	980K	Optimization Problem	
thermomech_dK	SB	200K X 200K	20M	Thermal Problem	
offshore	SC	250K X 250K	4.2M	Electromagnetics Problem	
oh2010	SD	366K X 365K	1.7M	Undirected Weighted Graph	
mario002	SE	389K X 389K	2M	Duplicate 2D/3D Problem	
pa2010	SF	421K X 421K	2.0M	Undirected Weighted Graph	
largebasis	SG	450K X 450K	5.2M	Optimization Problem	
fl2010	SH	484K X 484K	2.3M	Undirected Weighted Graph	
delaunay_n19	SI	524K X 524K	3.1M	Undirected Graph	
mc2depi	SJ	525K X 525K	2.1M	2D/3D Problem	
ecology1	SK	1M X 1M	4.9M	2D/3D Problem	
debr	SL	1M X 1M	4.1M	Undirected Graph Sequence	
roadNet-PA	SM	1M X 1M	3.0M	Undirected Graph	
thermal2	SN	1.2M X 1.2M	8.5M	Thermal Problem	
atmosmodd	SO	1.2M X 1.2M	8.8M	Computational Fluid Dynamics	
$belgium\_osm$[2]	SP	1.4M X 1.4M	3.0M	Undirected Graph	
g3_circuit	SQ	1.5M X 1.5M	7.6M	Circuit Simulation Problem	
transport	SR	1.6M X 1.6M	23.4M	Structural Problem	
kkt_power	SS	2M X 2M	12.7M	Optimization Problem	
netherlands_osm	ST	2.2M X 2.2M	4.8M	Undirected Graph	
curlCurl_4	SU	2.3M X 2.3M	26.5M	Model Reduction Problem	

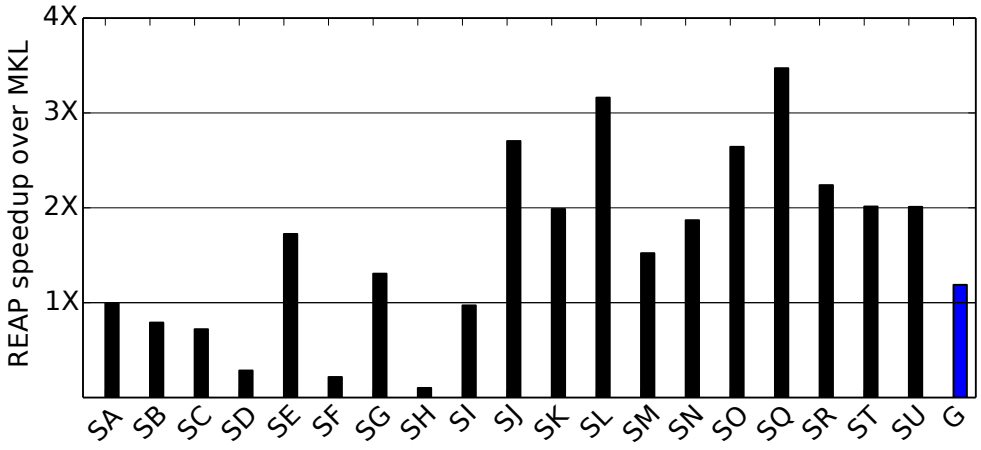
Design	Float-CSR	Float-DIA	Float-ELL	Int16-CSR	Int16-DIA	Int16-ELL	Int8-CSR	Int8-DIA	Int8-ELL
SpMV	1.18	1.25	1.87	1.68	1.87	2.93	2.20	1.86	3.42
SpGEMM	1.00	1.06	1.27	1.02	1.10	1.30	1.03	1.12	1.31

Table 2.4: REAP speedup compared to a 16-core Intel MKL for SpMV and SpGEMM for three sparse formats and different input precisions. We report the geometric mean of all the benchmarks in Table 2.3. The configurations are shown in the following format: precision-sparse format.

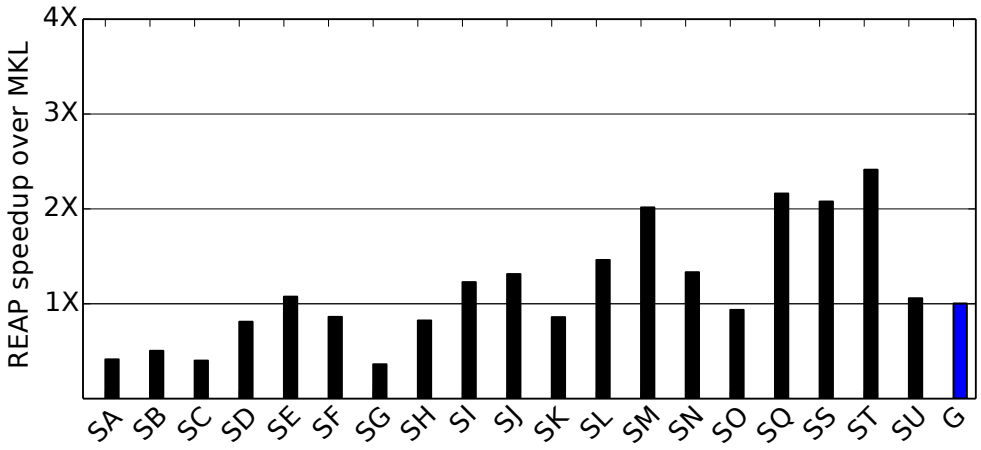
FPGA Baselines. We compare REAP with recent FPGA accelerators for SpMV and SpGEMM. HiSparse [35] is a high-performance SpMV accelerator designed for a multi-die HBM-equipped FPGA device. Similar to REAP, they build their prototype using the Xilinx HLS toolchain. We compare REAP with HiSparse using the same set of benchmarks as it is used in their evaluation. HiSparse needs a preprocessing step to transform the sparse data into an HBM-friendly format that we include its execution time when comparing with REAP. For SpGEMM, we compare REAP with SPAGHETTI [54], which is an FPGA accelerator for highly sparse matrices. Similar to SpMV, we evaluate the speedups using the benchmarks used in their work and some of them can be found in Table 2.3. SPAGHETTI utilizes a pattern-aware software scheduler that analyzes the sparsity pattern of the sparse inputs and schedules row-col pairs of the inputs for the FPGA. The authors did not report the execution time for the scheduler. Thus, we only compare the FPGA execution times. Unlike REAP, they do not overlap the CPU and the FPGA computation in their design.

2.5 Experimental Evaluation of REAP for SpMV and SpGEMM

Performance evaluation for various sparse formats and data precisions. REAP supports various sparse formats with a single FPGA design by modifying only the CPU's part of the design. REAP also supports different data precisions. Table 2.4 provides a summary of the REAP speedups compared to a 16-core Intel MKL running on a CPU listed in Table 5.2. We report the geometric mean of the speedups for all the benchmarks listed in Table 2.3. For all the configurations, REAP performs similarly or better than MKL. Unlike



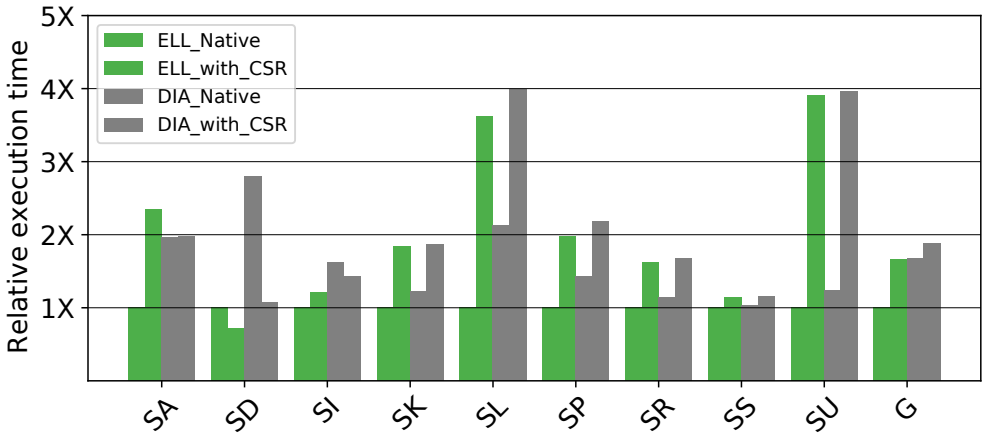
(a) SpMV



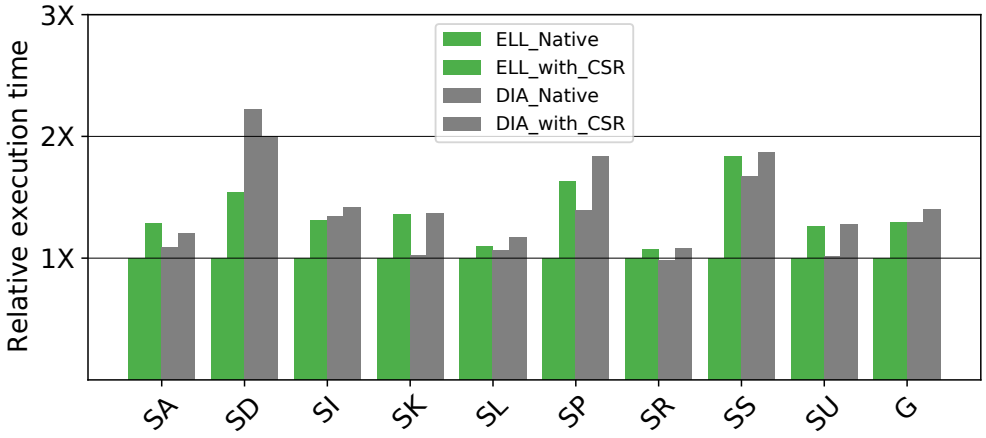
(b) SpGEMM

Figure 2.10: Speedup of REAP compared to 16-core Intel MKL for the SpMV and SpGEMM kernels with single-precision inputs stored in the CSR format. The last bars (blue bars) in each figure show the geometric mean of all the sparse matrices in Table 2.3.

MKL, which is primarily optimized for CSR format, REAP is optimized for various sparse formats. REAP also has advantages over MKL for low-precision data inputs. MKL only supports a limited number of data precision (*e.g.*, single-precision and double-precision inputs). For SpMV, there is up to 2× performance difference between the higher precision inputs (*e.g.*, *Float*) and lower precision input (*e.g.*, *Int8*). For SpGEMM, the RIR bundles include the coordinates and the values, limiting the performance gain for lower precision inputs. The main advantage of REAP over MKL is its ability to decouple the sparse data



(a) SpMV

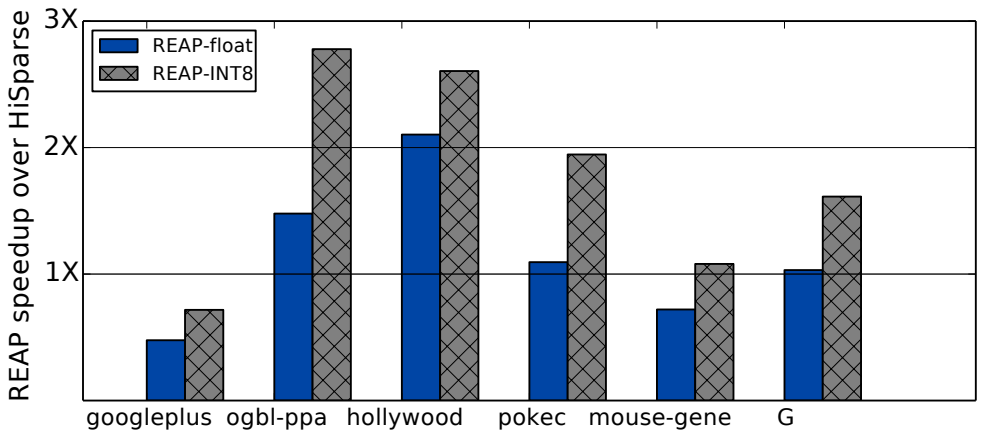


(b) SpGEMM

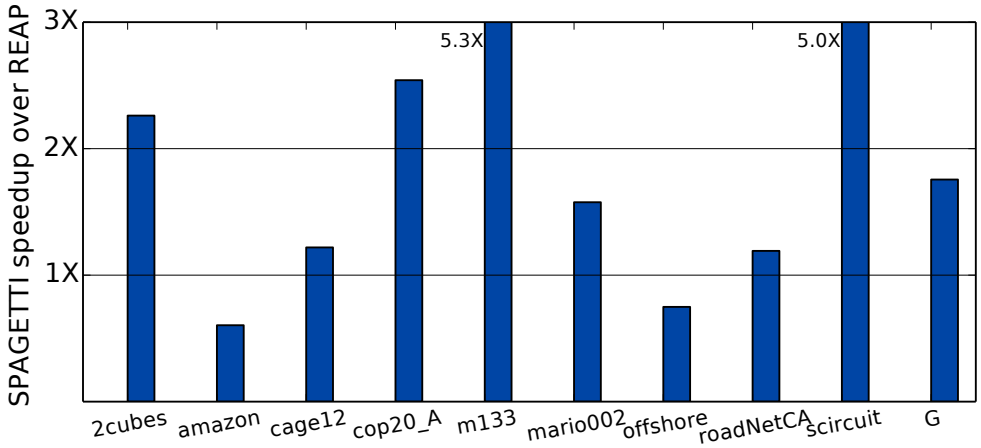
Figure 2.11: A comparison of the end-to-end execution time of two implementations of ELL and DIA formats, one that performs computation directly using ELL and DIA and the other converts the input from ELL or DIA to CSR format then performs the computation.

regularization from the numeric computation on the data and overlap the two computations using a CPU-FPGA system. REAP can also perform the computation directly using the original sparse format used to store the inputs, without requiring a preprocessing step to transform the input to a target format (*e.g.*, CSR for MKL).

Performance evaluation of SpMV. Figure 2.10a shows the speedup of REAP's SpMV design compared to 16-core Intel MKL on the CPU. For this experiment, the values are single-precision, and the input matrix is stored in a CSR format. The matrices are sorted based on their dimension, from small to large. REAP is 18% faster than MKL on average.



(a) SpMV



(b) SpGEMM

Figure 2.12: (a) Comparing REAP Float and Int8 SpMV implementation with HiSparse [35] for five graph processing benchmarks. (b) Comparing REAP's FPGA execution time with SPAGHETTI [54]. The last bar (G) shows the average for all benchmarks.

For large matrices, REAP outperforms MKL by up to 3×. It is easier to overlap the CPU and the FPGA computation for large matrices with more steps.

Performance evaluation of SpGEMM. Figure 2.10b shows the speedup with REAP for SpGEMM compared to 16-core Intel MKL. Similar to SpMV, both input matrices are stored in CSR format. REAP and MKL perform similarly overall for the CSR format. For the same reason as SpMV, REAP is up to 2× faster than MKL for larger matrices. REAP also performs better for matrices with a more irregular pattern of non-zeros. MKL's vectorization techniques are more effective for more simple sparsity patterns such as banded

matrices (*e.g.*, SO and SK). However, REAP's CPU reorganization outperforms the MKL's optimization for more irregular access patterns (*e.g.*, SS and ST).

Performance evaluation for DIA and ELL formats. Figure 2.11 compares the relative execution time for ELL and DIA formats. We compared two versions of each format. One version performs the computation directly using their original format (*i.e.*, *Native-ELL* and *Native-DIA* in Figure 2.11). This is similar to the method used in REAP. The other version converts the input to a CSR format and then performs the computation on the reformatted data. Overall, for DIA and ELL formats, the first method (REAP's approach) is faster than the second method. In particular, for the banded matrices (*e.g.*, SU), performing the computation directly on a DIA format significantly outperforms the other approach (transforming the input into CSR first). However, for the matrices with more irregular non-zero patterns (*e.g.*, SD and SI), the performance improves if the matrices are first transformed from DIA format into a more compact format like CSR before doing the computation. The same observation applies to the ELL format.

Performance evaluation compared to other FPGA accelerators. We compared REAP with state-of-the-art FPGA designs for SpMV and SpGEMM. Figure 2.12a shows REAP speedup compared to HiSparse [35] for the benchmarks used in their paper. HiSparse transforms the sparse inputs to an HBM-friendly format using a preprocessing step. Unlike REAP, their preprocessing stage is more expensive than the CPU's process in REAP, and it is not overlapped with the computation on the FPGA. Since the dataset includes mostly the graph dataset, we compared both Float and Int8 versions of REAP with their design that only supports Float. While their FPGA design enjoys higher bandwidth memory than REAP using an HBM memory, REAP performs 3% and 60% better than HiSparse for Float and Int8 versions, respectively.

Figure 2.12b shows the relative FPGA execution time of REAP and SPAGETTI [54]. Their design uses the software to analyze the sparsity pattern of the inputs and organize the non-zero before starting the computation on the FPGA. Figure 2.12b only compares the

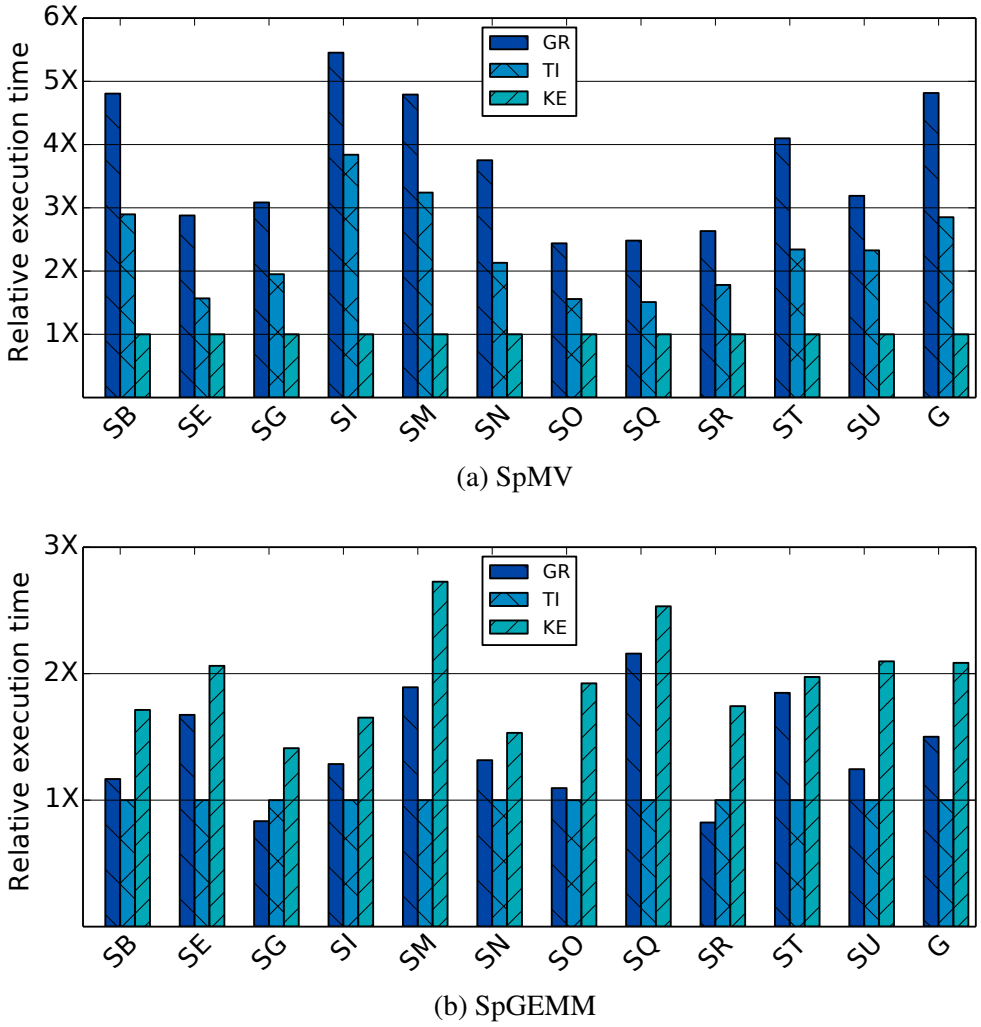


Figure 2.13: REAP execution breakdown for (a) SpMV and (b) SpGEMM. REAP execution time consists of generating RIR bundles(GR), transferring RIR bundles to FPGA (TI), and FPGA kernel execution(KE).

FPGA execution times for REAP and SPAGHETTI. They have not reported the software analysis execution time to allow us to compare the overall performance of the two systems. However, we suspect that their analysis execution time exceeds the data reorganization step in REAP. Besides, unlike REAP, they do not overlap the CPU and FPGA executions. On average, the FPGA execution time for SPAGHETTI is 1.75× faster than REAP.

REAP performance characterization. Figure 2.13 shows the breakdown of the REAP execution time for SpMV and SpGEMM designs. REAP's pipeline consists of generating RIR bundles(GR), transferring RIR bundles to FPGA (TI), and FPGA kernel execu-

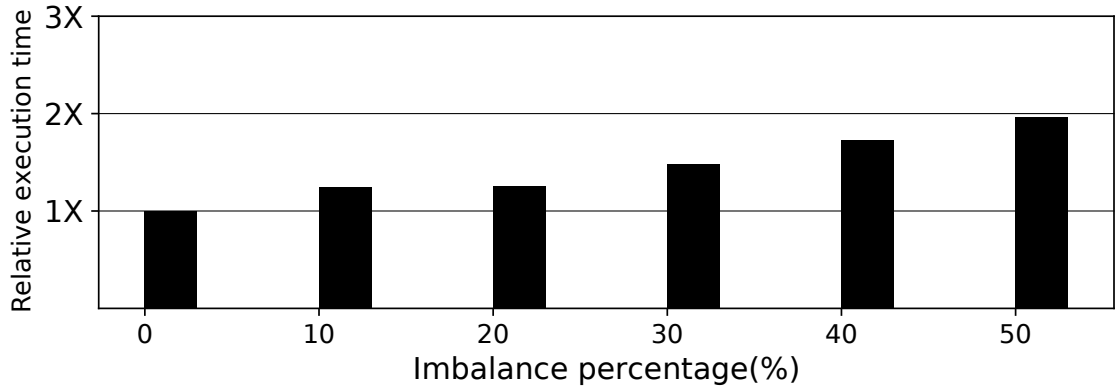
tion(KE). For SpMV, generating the RIR bundles exceeds the FPGA kernel execution. In contrast, for SpGEMM, the FPGA execution time surpasses the execution time of the CPU. For SpGEMM, the RIR bundles transfer time from the CPU to the FPGA memory is as expensive as generating the RIR bundles. For SpGEMM, the RIR bundle includes the value and indices from two matrices. Another important observation from Figure 2.13a is that the difference between the CPU and FPGA execution times is more significant for matrices with irregular access patterns (*e.g.*, SI, SM, ST) compared to matrices with regular access patterns (*e.g.*, SO, SR). In other words, for matrices with an irregular pattern of non-zero, the CPU requires more time to organize and schedule the non-zero for the FPGA. In contrast, the FPGA execution time merely depends on the overall non-zero operations.

Performance analysis of load imbalance in SpMV and SpGEMM.

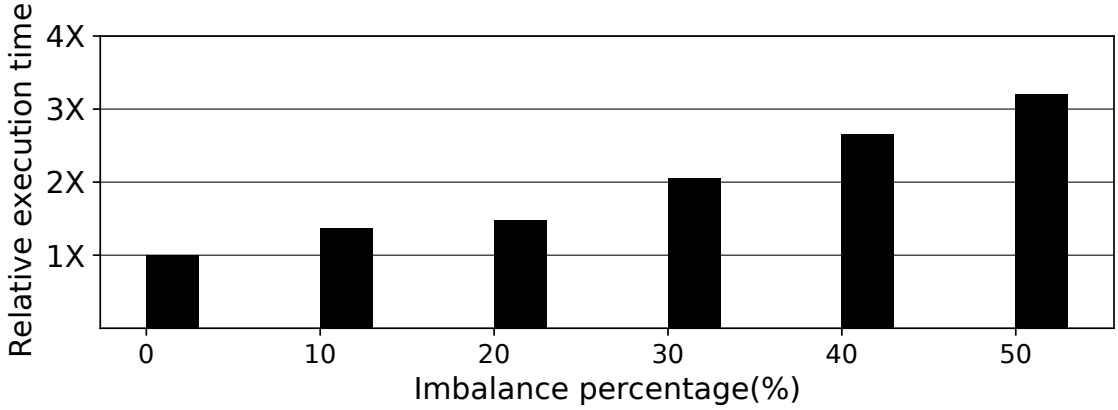
Figure 2.14 studies the impact of load imbalance on the FPGA execution time for SpGEMM and SpMV. We used the metric defined in Equation 3.1 to quantify the load imbalance ratio [33].

$$imbalance\_percentage = \frac{maximum\_work - average\_work}{maximum\_work} \times \frac{n}{n-1} \tag{2.1}$$

For this experiment, we used a synthetic benchmark with a dimension of one million and sixteen million non-zeros. We adjusted the imbalance ratio for each experiment while the total number of operations remained the same for all the experiments. There are up to 2×, and 3× differences in performance across different imbalance ratios for SpMV and SpGEMM, respectively. The load imbalance impacts SpGEMM execution times more than the SpMV design due to more expensive operations in the merge step. The load balance can possibly be improved using a dynamic and complex scheduling mechanism than the static round-robin scheduling used in REAP. However, using a more complex scheduling mechanism can increase the CPU execution time. Thus, it is unclear if the overall execution



(a) SpMV



(b) SpGEMM

Figure 2.14: Studying the impact of load imbalance on the FPGA execution time for SpGEMM and SpMV.

time can improve using other scheduling techniques. We leave studying the load balance problem and the scheduling techniques for sparse kernels for our future study.

2.6 Related Work on Accelerators for SpMV and SpGEMM Kernels

There is a large body of work on accelerating a variety of sparse linear algebra kernels on various platforms, including CPUs, GPUs, FPGAs, and ASIC designs. This section presents works aimed at SpMV and SpGEMM kernels.

General Purpose Architectures for SpMV and SpGEMM. There are highly optimized sparse linear algebra kernels for CPUs [3] and GPUs [1, 2]. In addition, there is a

large body of work on building cache-friendly and distributed algorithms [5, 14, 16, 46, 122], system level optimizations with parallelism on modern general purpose hardware including GPUs [79, 80, 88, 106, 136, 140]. TACO [22] addresses the problem of accelerating sparse computation with diverse formats by delegating the problem to the compiler. REAP's design provides the flexibility of TACO while exploiting FPGA to accelerate the computation.

FPGAs for SpMV and SpGEMM. Prior FPGA designs have been tailored to either SpMV [34, 38, 118] or SpGEMM [54, 77, 84, 119, 141]. Table 3.1 qualitatively compares REAP with some of the most recent FPGA accelerators for SpMV and SpGEMM.

For SpMV, the main challenge is supporting multiple random accesses to the dense input vector necessary to keep the FPGA units active. One solution is to replicate the dense vector in on-chip FPGA memory [66, 147]. This approach does not scale to large matrices due to limited on-chip memory. An alternative solution is to explore a custom sparse format to store the sparse input [18, 34, 35, 38, 43, 66, 118]. This needs a reformatting phase that transforms the data from its original format to the custom format. Unlike REAP, this reformatting step is performed before the computation begins. Thus, it is not overlapped with the computation on the FPGA. Recent work also utilizes FPGAs with high bandwidth memory (HBM) to accelerate the memory-bound SpMV kernel [35, 118]. [35] proposes a sparse format tailored for HBMs to increase the memory bandwidth utilization results compared to the conventional sparse formats like CSR. Like REAP, they split the kernel into multiple units and map each unit to different SLR regions to improve their design frequency. Unlike REAP, the processing time for arranging the sparse data into an HBM-friendly format is significant.

For SpGEMM, researchers have explored using 3D-stacked memory [152] and CAM-based accelerators [141]. Although feasible for small input matrices, they likely will not work with large inputs. Further, they are simulation-based studies without end-to-end system measurements on a realistic FPGA. Recent work used streaming accelerator for differ-

Table 2.5: Qualitative comparison of REAP with prior work that use FPGAs to accelerate sparse linear algebra kernels. [1] SpMM: Sparse-Matrix Dense-Matrix multiplication. [2] ACF: Accelerator-customized sparse format.

Accelerator	Implementation	Sparse Kernel	Sparse Format	Memory technology	Data precision
Sextans [119]	HLS	SpMM[1]	ACF[2]	HBM	Float
SPAGHETTI [54]	Chisel	SpGEMM	CSR/CSC	DRAM	Multiple precisions
HiSparse [35]	HLS	SpMV	ACF[2]	HBM	Float, Fixed-point
Serpens [118]	HLS	SpMV	ACF[2]	HBM	Float
REAP (this work)	HLS	SpMV, SpGEMM	Multiple formats	DRAM	Multiple precisions

ent variations of sparse matrix multiplication [54, 119]. Like REAP, SPAGHETTI [54] is a streaming accelerator that targets the multiplication of two highly sparse matrices. Similar to REAP, they use a software-hardware approach where a software scheduler analyzes the sparsity pattern and schedules the non-zeros of the inputs for the hardware. Unlike REAP, the work done by the software is not overlapped with the computation on the hardware. Besides, unlike REAP, which uses a row by row formulation for SpGEMM, they employed an outer-product formulation for their SpGEMM design. Their method suits better for applications where the sparsity pattern of the inputs does not change often. SEXTANS [119] is another recent streaming accelerator. Unlike REAP and SPAGHETTI [54], they target matrix-multiplication when only one of the inputs is sparse. They partition the inputs into smaller patches that fit on the on-chip memory of the FPGA. They limit all random memory accesses to FPGA's on-chip memory. The inputs in the off-chip memory are streamed in/out in batches and are always sequential. Unlike REAP, they schedule the non-zero in an out-of-order fashion. A recent work [9] characterizes the performance implications of different sparse formats in sparse workloads. Their study considered various metrics, including decompression overhead, latency, load balance, memory bandwidth utilization, and resource utilization. Similar to REAP, their result suggests the important role of sparse formats in the overall performance of the sparse kernels.

ASICs for SpMV and SpGEMM. Attaining good performance with FPGAs has been a challenge with sparse computation. Hence, there are many ASIC design proposals to accelerate SpGEMM and SpMV [53, 89, 97, 101, 120, 144, 148].

Extensor [53] is an ASIC that supports high-dimensional sparse data (*i.e.*, tensors).

Mapraptor [120] and Gamma [144] are both ASICs that employs a row-by-row formulation of SpGEMM similar to our work, which can be more efficient compared to ASIC that use an outer-product formulation [97, 148]. Similar to REAP, SMASH [61] uses a software-hardware approach to accelerate a variety of sparse linear algebra kernels. In Smash, they use the software to encode the sparse data as a *hierarchy of bitmaps*. This encoding is then used by an ASIC to avoid unnecessary memory accesses and reduce the overhead. Another recent work enhanced a tensor processing unit chip to support various sparse formats [100]. The high-level insight of their work is to decouple the compressed format used for tensor storage (they call it MCF) from the format used by the accelerator to perform the computation (they refer to it as ACF). Hence, they extend their sparse tensor accelerator with a hardware module that serves as a library to convert different MCFs to ACFs. These extensions require adding complex units such as prefix sum and parallel divide units. In contrast, in REAP, the hardware side always uses a fixed sparse representation (*i.e.*, RIR) that simplifies the hardware design and requires minimum metadata preprocessing by co-locating the input tensor values.

2.7 Summary

This chapter makes a case for a cooperative solution involving the CPU and the FPGA to accelerate sparse computation such as SpMV and SpGEMM. The CPU preprocesses the data and provides FPGA with regular computation, which is accelerated using a systolic-style design with logic units and on-chip memory. Decoupling the work done by the CPU and FPGA in a coarse-grained fashion enables us to overlap their execution for higher performance and support various sparse formats and data precisions. The speedups over mainstream libraries with our end-to-end system demonstrate that it is possible to achieve performance with FPGAs while being flexible to adapt to various formats.

CHAPTER 3
AN ASIC ACCELERATOR FOR SPARSE CONVOLUTIONAL NEURAL NETWORKS

In Chapter 2, we presented a synergistic hardware/software system that increases the performance and versatility of sparse kernels with input densities under 1% and inputs of varying sparsity patterns. There is another class of sparse problems that has higher input densities and fixed sparsity patterns. Sparse convolutional neural networks are a notable example of these problems. This chapter presents a novel hardware accelerator for the inference task of sparse convolutional neural networks (CNNs). To build a hardware accelerator for sparse CNNs, we first formulate the convolution operation as general matrix-matrix multiplication (GEMM) using a transformation called Image to Column (IM2COL). Then we build a hardware unit to perform IM2COL transformation of the input feature map coupled with a systolic array-based GEMM unit. Our accelerator for sparse CNNs inference, which we call **SPOTS** carefully overlaps the IM2COL transformation with the GEMM computation to maximize parallelism. We propose a novel design for the IM2COL unit that uses a set of distributed local memories connected by a ring network that improves energy efficiency and latency by streaming the input feature map only once. The systolic array-based GEMM unit in the accelerator can be dynamically configured as multiple GEMM units with square-shaped systolic arrays or as a single GEMM unit with a tall systolic array. This dynamic reconfigurability enables effective pipelining of IM2COL and GEMM operations and attains high processing element utilization for a wide range of CNNs. Further, our accelerator is sparsity-aware, improving performance and energy efficiency by effectively mapping the sparse feature maps and weights to the processing elements, skipping ineffectual operations, and unnecessary data movements involving zeros. After the training phase, we apply a group-wise pruning followed by a preprocessing step

to reformat the pruned weight into a hardware-friendly sparse format *prior* to the hardware execution. The non-zero weights pattern does not change throughout the inference task, so the approach is effective for sparse CNN inference.

3.1 Overview of SPOTS

Neural networks are widely used in numerous domains such as video processing [68], speech recognition [23], and natural language processing [50, 113]. They have surpassed or are close to human accuracy in many of these tasks. To achieve such accuracy, the training phase involves large datasets and several iterations of weight updates, which can take several hours or days to complete. Hence, the training phase is typically performed in the cloud or on a large cluster of machines. Unlike training, the inference is performed both in the cloud as well as at the edge devices (e.g., mobile devices or the Internet of Things (IoT) devices). It is often desirable to compute on edge devices, especially when network connectivity is limited or unavailable. The edge devices typically have limited memory and compute resources with strict requirements on energy usage. Thus, our hardware design targets the CNN's inference task on edge devices.

A number of applications, including image processing, use convolutional neural networks (CNNs). CNNs can have multiple layers, including convolution layers, fully connected layers, and pooling layers, with most of the computation taking place in the convolution layers. A convolution operation involves sliding a smaller filter window over an input array with a stride size, producing patches. A CNN layer has multiple features: the number of filters, kernel size, stride size, and channel size. Thus, designing an accelerator that performs well for all types of layers in a CNN is challenging given the wide range of features. Further, supporting sparse inputs introduces additional complexity to the design.

Given the importance of CNNs in various applications, numerous CNN accelerators have been explored by the community [8, 19, 21, 30, 36, 40, 42, 48, 56, 81, 98, 102, 103, 109, 145]. Often, designs are tailored to particular CNN architectures [48, 139]. Hence,

they suffer from low resource utilization for certain layer shapes and sizes. With respect to sparsity-awareness, many prior approaches handle sparsity in either the weights [69, 145] or the input feature map [4, 8]. Many recent designs support sparsity in both the feature map and weights [30, 42, 98, 102]. As an example, SparTen [42] uses a costly prefix sum unit to locate the non-zero pairs that match. Their sparsity-awareness method for finding the non-zero pairs contributes to 42% and 62% of the total area and energy, respectively. Sparse-PE [102] avoids the need for expensive hardware units for finding non-zero pairs by decompressing the sparse vectors into a dense format before locating them. However, this solution involves an additional decompression step (zero insertion) and uses large buffer sizes inside each core in order to store the vectors densely. Unlike the other two methods, SCNN [98] uses a Cartesian product method to avoid the index matching phase altogether. The main drawback of such an approach is that it introduces irrelevant partial products. Finally, while prior sparsity-aware accelerators, including SparTen and Sparse-PE, successfully prevent the ineffectual multiplications (multiplications involving zeros), they fail to avoid many unnecessary data transfers. The index matching phase requires the sparse vectors to be fetched by each processing element (core) even if they do not contribute to any output result (*e.g.*, when they are not matched with any non-zero).

To implement CNNs, one way would be to realize a convolutional layer as a large, single General Matrix-Matrix Multiplication (GEMM) using a data reorganization transformation called **Image-to-Column (IM2COL)**. Unsurprisingly, many mainstream frameworks use this approach since highly optimized GEMM primitives are available (*e.g.*, BLAS [13] or CuBLAS [94]). One method to accelerate the convolution computation is to offload only the GEMM operation to a hardware accelerator. However, the IM2COL operation accounts for a sizable fraction of the execution time (29% of the total time). Additionally, IM2COL performs many redundant memory accesses, which contributes to the overall energy consumption. Furthermore, offloading only the GEMM operation to a hardware accelerator and doing the IM2COL operation in software prevents fine-grained pipelining of

the IM2COL transformation and the matrix-matrix multiplication operation. Finally, performing the IM2COL operation in hardware reduces the amount of data transfer between the CPU and the hardware accelerator.

We propose a dedicated hardware IM2COL unit that operates in parallel with the hardware GEMM unit. Using this specialized IM2COL unit eliminates the redundant data accesses and enhances the data reuse, allowing us to improve performance and reduce energy consumption. A novel aspect of the IM2COL unit in SPOTS is that it has a collection of patch units (PUs) that streams the input only once, performs data reorganization, creates multiple patches in parallel, and eliminates redundant accesses. To eliminate redundant accesses, each patch unit in the IM2COL unit has three local buffers that identify overlapped elements between patches and avoid expensive DRAM accesses. These patches are subsequently fed into a systolic array-based GEMM unit.

The GEMM unit in SPOTS is efficiently pipelined with the IM2COL unit. The GEMM unit can be configured as a single tall-thin unit or multiple GEMM units with square-shaped systolic arrays with processing elements (PEs). The tall-thin shape better balances the memory bandwidth requirement of the GEMM unit and the throughput of IM2COL unit, which allows efficient pipelining of operations between the PEs performing the matrix multiplication with the PUs executing the IM2COL reorganization. The dynamic reconfigurability of the GEMM units enables SPOTS to achieve high PE utilization with a variety of convolutional layers varying in number of filters, kernel size, stride values, and feature map dimensions. In addition to convolutions and fully connected layers, SPOTS also supports pooling layers via minor enhancements to the IM2COL unit.

One of the most important features of SPOTS is that it supports sparse inputs for both weights and feature maps. Sparsity in weights results from the pruning step in CNNs. Pruning reduces computation and memory footprint by eliminating weights after training without affecting accuracy. SPOTS uses sparsity to skip data transfer and computation for sparse regions. Our new sparse format, tailored to our group-wise pruning algorithm, re-

duces the storage requirements for the weights by a substantial amount in comparison to random pruning [48] while allowing PEs to access the weights with relatively high bandwidth. Finally, SPOTS tags and skips blocks of zeros in the result of the IM2COL unit and weights before entering the systolic array, saving computation cycles and memory transfers. Further, this approach helps SPOTS avoid the potential load imbalance caused by an uneven distribution of the zeros in the inputs since the zero blocks are skipped for *all* PEs.

3.1.1 Novelties of SPOTS

- We design a novel IM2COL design that executes in parallel with the GEMM unit and minimizes the latency and DRAM accesses by exploiting the data locality that exists in the patches.
- We propose a dynamically reconfigurable GEMM unit with the ability to adapt to different CNN layers and shapes to achieve high PE utilization.
- We present a sparsity-aware design that reduces storage and computation requirements by exploiting sparsity in both feature maps and weights.

3.2 Background on CNNs, IM2COL, and Sparsity-Awareness in CNNs

We provide background on CNNs (Section 3.2.1), structuring the convolution operation as general matrix-matrix multiplication with the help of the IM2COL transformation (Section 3.2.2), and leveraging sparsity to improve performance and energy efficiency (Section 3.2.3).

3.2.1 Convolution Neural Networks

A Convolution Neural Network (CNN) consists of a series of layers. In a CNN, each layer extracts a high-level feature from the input data, called a *feature map* (fmap). There are different types of layers in CNNs, including convolution, activation (*e.g.*, non-linear

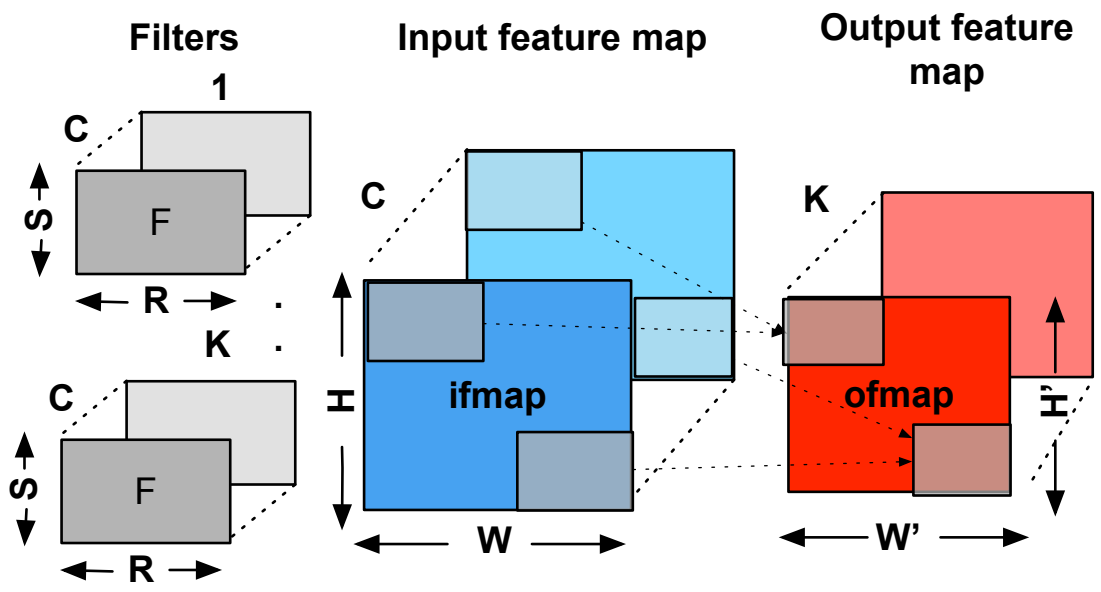
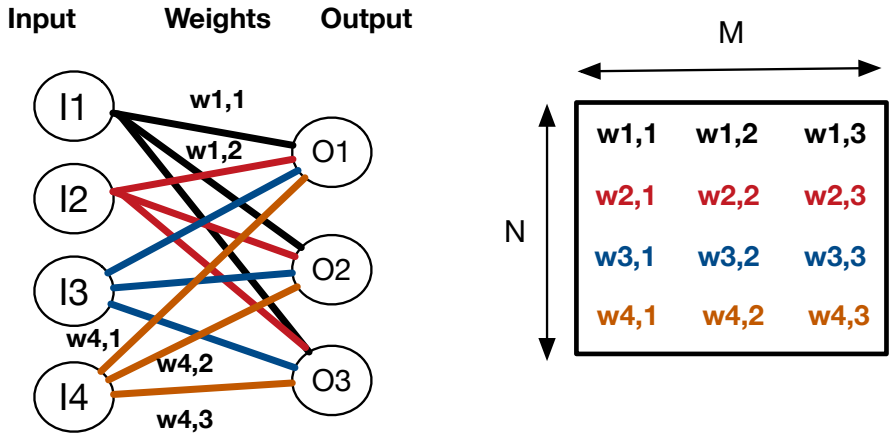


Figure 3.1: Illustration of a convolution layer along with its inputs.

operator), pooling, and fully connected layers. The convolutional layers are the main layers in a CNN. They are responsible for the bulk of the computation. There are several filters in each convolutional layer. The values of these filters (i.e., weights) are learned during the training phase. In the inference phase, the network classifies new inputs presented to the network.

Figure 3.1 shows the computation in the convolution layer. The input feature map is structured as a 3-D tensor with W, H, and C as its width, height, and the number of channels, respectively. Similarly, the filters are structured as 3-D tensors with width (R), height (S), and C channels. The filters and the input feature maps have the same number of channels. There are K filters in this example. Typically, a collection of N input feature maps are convolved with K filters (*i.e.*, a batch size of N). For inference tasks, it is common to use a batch size of 1. For some convolution layers, a 1-D scalar bias is also added to the result, which is not shown in Figure 3.1.

Fully Connected (FC) Layers. Most CNNs have one or more *fully connected* layers at the end of the network to extract dense features from the input vector. In a fully connected



(a) Fully connected layer connections between the input and output neurons

(b) The matrix representation of the weights in an FC layer

Figure 3.2: An FC layer and how the weights between the input and output neurons can be transformed into a matrix before performing a matrix-vector multiplication. The labels are shown only for some of the weights.

layer, all the inputs neurons are connected to all the outputs neurons in the next layer (see Figure 3.2(a)). Figure 3.2(b) shows how the weights between the input and output neurons are transformed into a matrix. The number of rows and columns in the matrix matches the number of input and output neurons in a layer. For the example in Figure 3.2, there are 4 input and 3 output neurons. FC layers can be computed using matrix-vector multiplication.

3.2.2 Transforming Convolution to General Matrix-Matrix Multiplication

The convolution operation can be transformed into general matrix-matrix multiplication (GEMM) using the IM2COL transformation. To structure a convolution operation as matrix-matrix multiplication, we need to create two matrices from two inputs: input feature map and the K filters. Figure 3.3 illustrates how the two matrices are created. The product of these two matrices is equivalent to the result of the convolution operation. For building the weight matrix, each filter is mapped to one row of the weight matrix. The weight matrix will have K rows if there are K filters in a leyer (Figure 3.3(a)). The number of columns in the weight matrix is $R \times S \times C$.

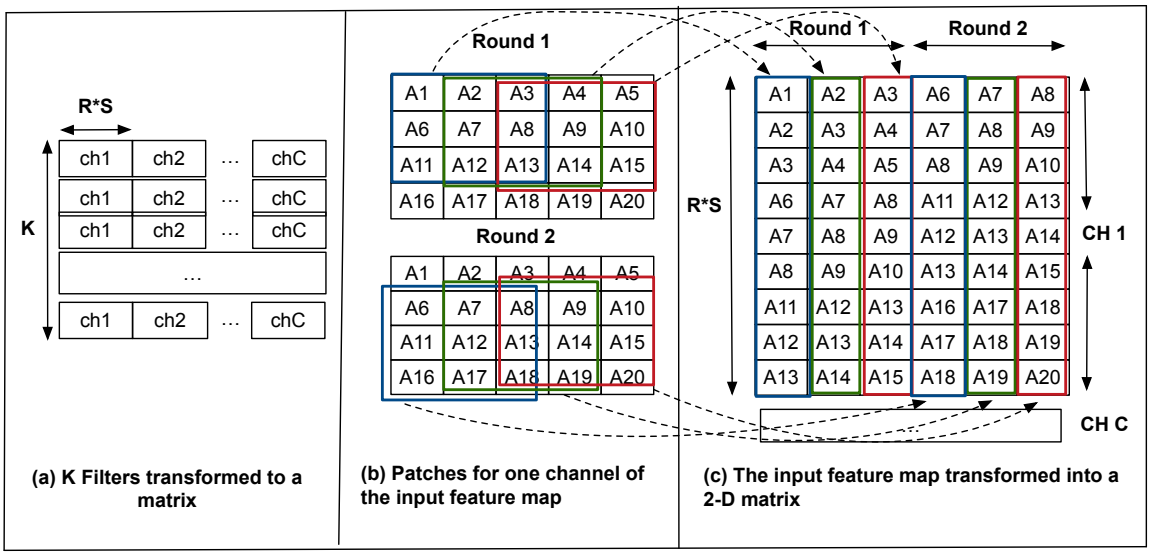


Figure 3.3: Transforming the inputs of a convolution layer (i.e., input feature map and filters) into two matrices to use a GEMM-based formulation of convolution.

To construct a 2-D matrix from a 3-D input feature map, a more complex transformation is required. This transformation is called **Image to Column (IM2COL)**. The IM2COL result depends on the kernel size and the stride size, which are the two parameters of the convolution. In convolution, each filter slides across different positions in the input feature map. We call all elements in the input feature map covered by the filter as a *patch* or a tile. Patches are often overlapped with each other when the stride size is less than the filter size. This overlap results in the repetition of the same element of the input feature map in multiple patches. Figure 3.3(b) and Figure 3.3(c) illustrates the IM2COL transformation with an example filter of size $(3 \times 3 \times C)$ and a stride of 1. Each column of the matrix produced by the IM2COL transformation corresponds to one patch where the filter is applied for all C channels, and it has $R \times S \times C$ rows. Figure 3.3 shows the patches for one channel. Finally, the product of the two matrices (Figure 3.3(a) and 3.3(c)) generates the output of the convolution operation.

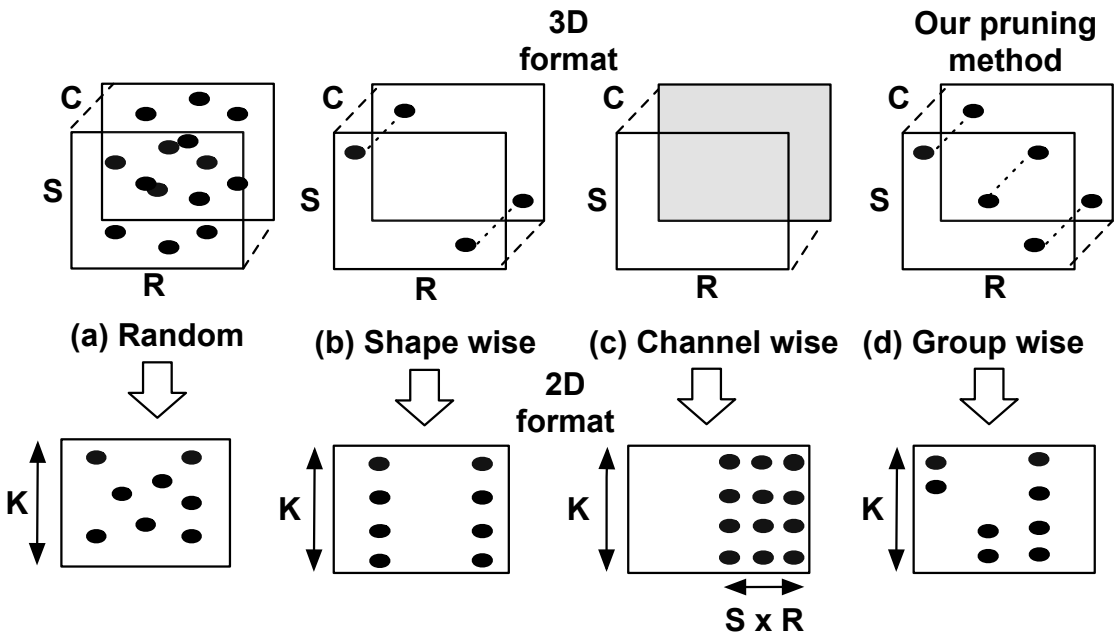


Figure 3.4: This figure shows the resulting zeros in the 2-D matrix representation of the filter while pruning the filters at different granularities and their corresponding matrix format. A dark dot indicates that the point is being pruned.

3.2.3 Sparsity-Awareness in CNNs

Given a layer in the CNN, a significant fraction of the values in the weights and the feature map values can be zeros. A pruning step is often applied to remove unimportant and redundant weights during the training phase, resulting in zeros in the final trained weights. Unlike the zeros in the weights that are known after the training phase, the input feature map can also have zeros that are not known until the inference task. A sparse format can be used to compress the pruned weights. In addition to reducing the model size, different hardware accelerators use sparsity to improve performance and energy efficiency of the design. The performance improvement comes from eliding multiplications and minimizing data movement when it involves zeros.

Techniques for pruning filters. Pruning is typically employed to increase the sparsity of the weights in the filters. There are two strategies for pruning: random pruning and structured pruning. The *random pruning* strategy sets a weight to zero if it is below a

threshold value [49]. Non-zero weights are typically stored in a compressed sparse format after the pruning step. However, sparse formats involve indirect accesses and require extra steps for extracting non-zero elements and matching indices. In contrast, the *structured pruning* strategy addresses irregular accesses due to random pruning [62, 69, 135]. A structured pruning method removes redundant weights only from certain locations or in a specific block size. Figure 3.4 shows pruning at different levels with various pruning methods. The dark points represent pruned weights in the filter. When we convert a 3-D filter to a 2-D representation using the strategy shown in Figure 3.3(a), the resulting zeros in the 2-D matrix are shown in the second row of Figure 3.4. The random pruning strategy results in an irregular pattern of zeros. A coarse-grained structure (*e.g.*, channel-wise) for pruning can result in a group of zero columns in the 2-D matrix, which is more hardware friendly. The downside is that it may compromise network accuracy. With a fine-grained structure (e.g., shape-wise or group-wise) you get closer to random pruning while having a regular structure with zeros. We will describe the details of our group-wise pruning in Section 5.1.

3.3 Motivation for a GEMM-based Formulation of Convolution and Sparsity-Awareness Design in SPOTS

In this section, we first discuss the inefficiencies of mapping a convolution operation to an array of processing elements the way that some of the prior work proposes. Then we discuss the advantages and disadvantages of using a GEMM-based formulation for convolution with an IM2COL transformation. Finally, we review some of the sparsity-aware proposals and their disadvantages to motivate SPOTS sparsity-aware design.

3.3.1 The Inefficiencies of Mapping Convolution Operation to Processing Elements

The sliding-window nature of the convolution operation introduces overlaps between the patches. This makes it difficult to map a convolution operation to a set of processing el-

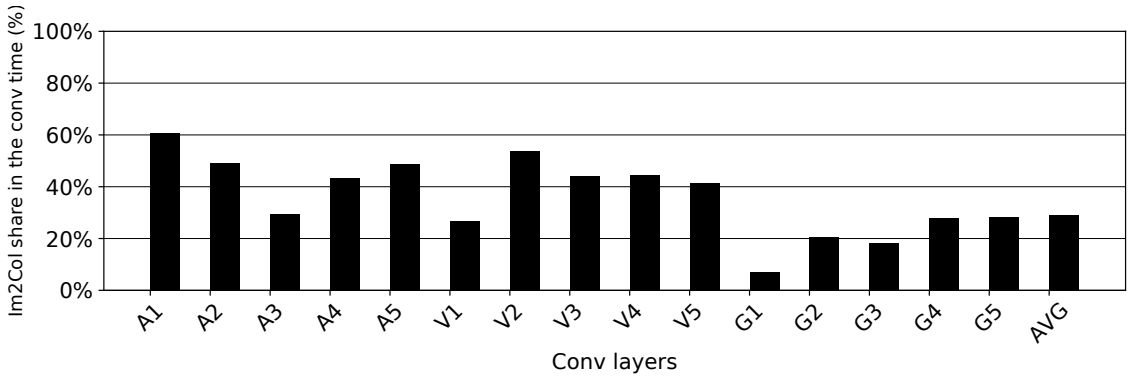


Figure 3.5: The percentage of the total execution time spent in the IM2COL transformation for various convolution layers from AlexNet, VGGNet, and GoogleNet for a CPU system.

ements (PEs) in a hardware accelerator. It is common to design a fetch unit within each PE to retrieve the input patch, communicate the patch with other PEs, and manage the partial results. For communication between PEs, a specialized interconnect is typically used. Prior work such as SCNN [98] and Eyeriss [21] adopt this approach. One of the main weaknesses of this approach is that the dataflow and interconnection networks are heavily customized for the convolution operation. Hence, both SCNN and Eyeriss are inefficient for other layers, such as fully connected layers. For example, SCNN can achieve 25% of the peak throughput for fully connected layers. Similarly, Eyeriss fails to achieve high PE utilization for small batch sizes.

3.3.2 Benefits and Challenges of Convolution with IM2COL

A separate IM2COL transformation allows the construction of input patches to be separated from the eventual computation performed on them. The IM2COL transformation can identify data overlap among different patches as each filter slides across different positions in the input feature map. Further, a separate IM2COL transformation can enable one to use highly optimized primitives or even available hardware accelerators for GEMM. However, performing the IM2COL transformation in software may not provide the best possible performance for the following reasons. First, a naive IM2COL transformation can result in numerous redundant memory accesses. Sliding the filters over the input feature map creates

numerous repetitions in the IM2COL patches. Depending on the filter size and stride size, the number of memory accesses can be $9\times$ more than the number of elements, indicating that many elements are accessed repeatedly. Second, IM2COL can account for more than 60% of the convolution operation's execution time. On average, the decoupled IM2COL transformation spends 29% of the overall execution time for various layers in AlexNet, VGGNet, and GoogleNet for a CPU system.

Section 3.4 describes our accelerator that performs IM2COL on-the-fly, extracts significant parallelism between various patches, and uses the hardware IM2COL unit to simplify the hardware accelerator for GEMM without the need for complex interconnection networks.

3.3.3 Drawbacks of Prior Sparsity-aware Designs

To motivate our design, we review some of the main drawbacks of prior sparsity-aware designs.

Redundant multiplications. One challenge of a sparsity-aware design is to find non-zero pairs to multiply depending on their indices. SCNN [98] uses a *Cartesian product* method to avoid the index matching step entirely. The Cartesian product of two vectors produces an output vector that includes the product of each element from the first vector with all the elements from the second vector. A Cartesian product's *all-to-all* nature eliminates the need for an additional step to match the non-zero values in two vectors. The major weakness of this approach is that it generates some unnecessary partial products during the multiplication phase that do not contribute to any final output.

Expensive hardware to find the non-zero pairs to multiply. Other proposals such as SparTen [42] and GoSPA [30] avoid redundant multiplications by using an index matching step, sometimes referred to as an *intersection*. This approach has two major drawbacks. First, the intersection step is often performed by using expensive hardware (*e.g.*, prefix sum in SparTen). Consequently, the intersection unit introduces significant area and energy

costs to a sparsity-aware design. For example, in SparTen, the sparsity handling contributes to 42% and 62% of the total area and energy of the design, respectively. Second, the intersection step is in the execution's critical path. Therefore, multiplication units may experience frequent idle cycles while waiting for the intersection results.

Extra decompression steps. Some methods like Sparse-PE decompress the sparse data into a dense format before finding the non-zero pairs. Using a dense format helps them simplify the hardware unit for index matching. However, it introduces a further step of decompression (e.g., zero insertion) and requires large buffer sizes to store vectors in a dense format, which reduces the benefits of using a compressed format.

Unnecessary data traffic. Methods used in SparTen and Sparse-PE are successful in avoiding redundant multiplications that involve zeros as a result of the index matching step. However, both designs still generate unnecessary data traffic. In the index matching phase, the sparse vectors must be fetched by each processing element (core), even if they do not contribute to the final result.

Custom routing needed for partial products. Sparsity-aware accelerators, such as SparTen, Sparse-PE, and SCNN, have two separate units, one for multiplication (*i.e.*, generating partial products) and one for accumulation (*i.e.*, adding the partial products to produce the final output). Due to the need to route the partial products from the multiplication units to the accumulation units, this approach adds complexity to the design.

Contrary to existing work on sparsity-aware designs, our aim with SPOTS is to eliminate redundant multiplications, minimize expensive hardware units, reduce metadata and memory footprints, and reduce data traffic, resulting in both performance and energy savings.

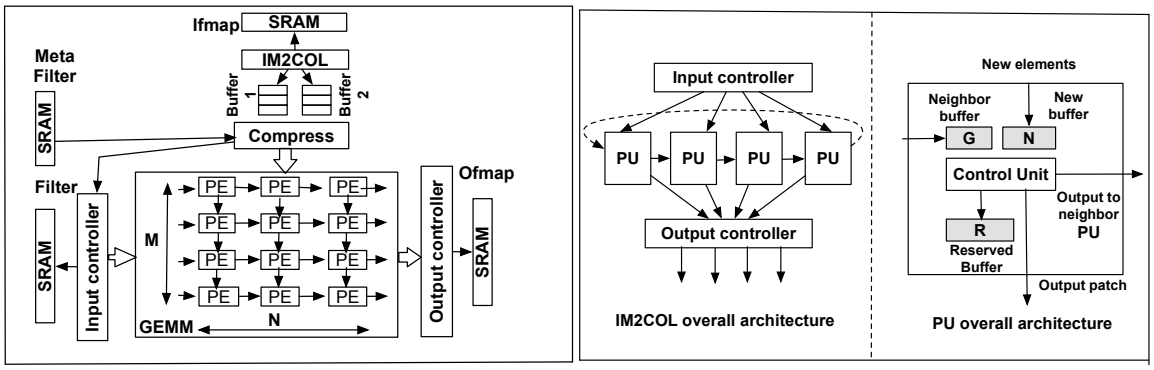
3.4 SPOTS: Our Hardware Accelerator for Sparse CNNs

This section describes our hardware accelerator for sparse CNN inference. Our design uses a GEMM-based formulation of a convolution operation. Our design goals are four-

fold: (1) significant performance and energy benefits, (2) support multiple CNN layers and efficiently execute various layers of varying shapes and sizes, (3) exploit the sparsity in the weights and the filters to reduce the sotrage and computation of CNN's inference, and (4) fine-grained pipelining of the IM2COL operation with the GEMM computation.

We propose a hardware unit for the IM2COL transformation that is synergistic and pipelined with the hardware unit for GEMM. The IM2COL unit reads the 3-D input feature map, and creates a set of linearized patches. The IM2COL unit consists of patch units (PUs) that are responsible for constructing linear patches. As values are streamed in, the PUs construct patches and forward the overlapped elements to neighboring PUs. Once a PU collects all the values in a patch, it sends the complete patch to the GEMM unit. As a result of this approach, the IM2COL unit only reads in values from the feature map once and reuses them, avoiding redundant memory accesses.

We design a dynamically reconfigurable GEMM unit with a systolic array-based design. The array of PEs can be configured as a tall array to balance the work between IM2COL and GEMM computation. GEMM units can be configured as small GEMM units (Section 3.4.4) to maintain a high PE utilization with CNN layers of varying shapes. Through dynamic reconfigurability, the computation of various shape CNN layers can be mapped to the PEs in the GEMM unit efficiently. Additionally, our design is sparsity-aware that allows it to detect and skip zeros in the inputs (Section 3.4.3). Figure 3.6a shows the overall architecture of our accelerator. The two main components are the unit for the IM2COL transformation and the GEMM unit. IM2COL and GEMM are connected by two buffers that enable effective pipelining of operations between them. The *compress* unit detects and skips the zero blocks in the feature map and weights before they are sent to the GEMM unit. Next, we will explain each component in more detail.



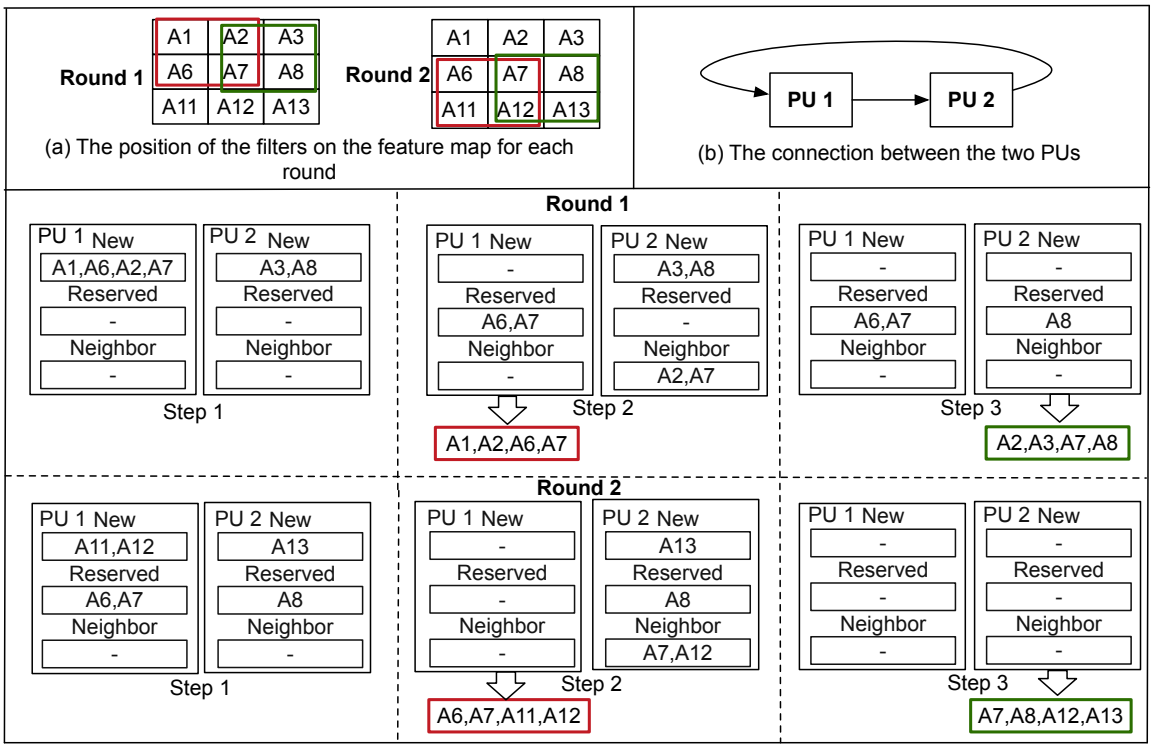
(a) Overall architecture of SPOTS (b) Overall IM2COL architecture and patch unit

Figure 3.6: (a) The overall architecture of our accelerator with the IM2COL unit and a systolic array-based GEMM unit. (b) The Overall IM2COL architecture and patch unit internals.

3.4.1 The IM2COL Unit

The IM2COL transformation creates a 2-D matrix from a 3-D input feature map, which reduces convolution to matrix multiplication (Section 3.2.2). The IM2COL transformation is challenging because it inherits some of the complexity of the convolution operation. The IM2COL has a complex memory access pattern that results in many redundant accesses to the memory.

To accelerate IM2COL and minimize the number of accesses to the input feature map elements, we propose a distributed hardware structure consisting of Patch Units (PUs)(Figure 3.6b). A key insight in our IM2COL unit is that we use the localities that result from the overlap between patches as we slide the filters over the input feature map both vertically and horizontally. PUs are responsible for building patches one at a time. Our design goal is to read the input feature map from SRAM only once. This is accomplished by each patch unit having small local buffers containing values that can be used to build future patches. PUs are also connected via a ring network, which enables them to communicate elements locally and avoid redundant accesses to the input feature map in SRAM. Figure 3.6b shows the overall architecture of our IM2COL unit that consists of three main components: input controller, PUs, and output controller.



(c) The steps to generate the patches with two PUs

Figure 3.7: Illustration of patch generation using the PUs in the IM2COL unit. We show an IM2COL unit with 2 PUs for exposition. (a) The input feature map with one channel. We show the sliding windows used to generate patches with a stride of 1. (b) The two PUs are interconnected by a ring network. (c) There are two rounds. Round 1 corresponds to patches belonging to the first row of sliding windows over the input feature map. Similarly, round 2 corresponds to patches belonging to the second row of sliding windows.

The input controller reads the elements of the input feature map from SRAM and forwards them to the appropriate PU units. Along with sending values from the input feature map to the respective PUs, the input controller maintains extra metadata. This metadata contains information about the position of the current patch. For some convolution layers, the stride size is the same as the kernel size. There is no overlap between the patches in those cases. In those situations, the input control forwards its output directly to the output controller by skipping the PUs.

The IM2COL unit contains multiple PUs. The PUs are the main components of the IM2COL unit for generating patches. The internals of the PU are shown in Figure 3.6b. Each PU has three buffers: the new buffer, the neighbor buffer, and the reserved buffer. The

new buffer (N) contains the newly fetched element from the input controller. The neighbor buffer (G) stores the elements received from the neighboring PU. The reserved buffer (R) stores some of the elements previously received at that PU in the previous rounds. We store the row and column indices (i.e., coordinates) along with the value for each element. Every PU has a control unit that manages the buffer and generates patches. The controller determines whether an element should be forwarded to the neighboring PU or maintained in the reserve buffer for future use.

Each patch has a unique identifier (i.e., the row and column of the top-left element). The control unit in a PU uses the patch identifier, the filter size, and the stride size to determine which elements need to be (1) fetched from the input feature map, (2) forwarded to the neighboring PUs, and (3) stored in the reserve buffer for future rounds. As an example, all elements must be accessed from the input feature map when the first patch is processed by a PU.

In a given round, all the elements required for adjacent patches are provided by neighboring PUs. A PU typically receives $K^2 - K \times S$ elements from the neighboring patches as long as it is not the first patch in a given round, where K is the size of the kernel and S is the stride size. We assign all patches that belong to the same column (i.e., column index of the top-left element) in different rounds to the same PU. Therefore, the PUs store some elements that may be useful to build patches in subsequent rounds in the reserved buffer. The process is repeated for all channels in the feature map.

The total number of elements that are overlapped between the vertical patches for a given filter size is $C \times W \times (K - S)$ where W is the width of the input feature map. This represents the maximum amount of data reuse possible with a reserve buffer. Further, the width and the channel size are inversely proportional to each other. For example, the first few layers of a CNN often have a small number of channels that are wider. In contrast, the later layers of the CNN have larger channels of smaller width. Therefore, a small reserve buffer can provide substantial data reuse even for larger layers. When the number

of overlapping elements between vertical patches exceeds the size of the reserved buffer, the input controller skips the reserved buffer and fetches the element from SRAM again. Data reuse in such cases is limited to horizontally adjacent patches.

Finally, the output controller organizes patches formed by each PU and manages communications with the GEMM unit. It coordinates double buffering that enables the IM2COL unit and the GEMM unit to execute simultaneously.

Figure 3.7 illustrates the process of generating the patches using the PUs in our IM2COL unit. For example, PU1 receives four elements (A1, A6, A2, A7) from the input controller and stores them in the new buffer in step 1. Similarly, PU2 receives two new elements (A3, A8). In a later step (i.e., step 2), the PU2 will receive other elements from the PU1.

Our hardware IM2COL unit offers two benefits: energy efficiency and performance. Accessing the smaller SRAM and performing integer operations (such as computing row and column indices) consumes significantly less power than accessing DRAM or large SRAMs. Therefore, our design provides significant energy savings. Further, our distributed collection of PUs unlocks extra parallelism beyond parallelism among the channels, allowing multiple patches to be built simultaneously by different PUs in the IM2COL unit that boosts performance.

3.4.2 The GEMM Unit

Our hardware unit for accelerating GEMM is a systolic array-based design. As opposed to previous proposals that used systolic arrays for GEMM [21, 69, 70], we add dynamic reconfigurability to the GEMM unit. The GEMM unit in SPOTS can be configured either as a tall-shaped systolic array (the height is considerably larger than the width) to maximize data reuse or as multiple GEMM units with square-shaped systolic arrays. Figure 3.8(b) illustrates our systolic array-based design for GEMM with a tall array.

The use of a tall systolic array-based architecture for GEMM has two main advantages. One of the inputs for the GEMM unit comes from the IM2COL unit. When a tall-shaped

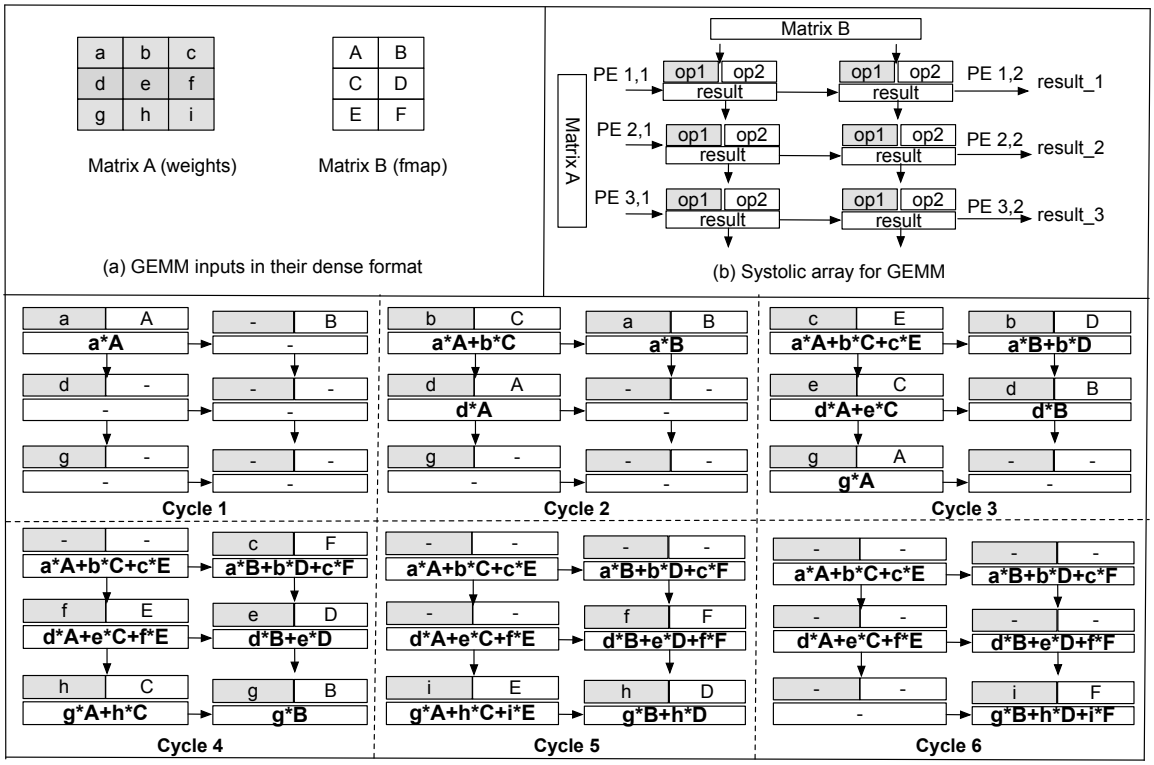


Figure 3.8: Illustration of our GEMM unit. (a) Inputs to the GEMM unit. (b) A tall array for the GEMM unit. (c) Illustration of GEMM computation at various steps. We show the current inputs and the partial results computed till a step for each PE. We demonstrate the output-stationary attribute of our design.

array is used, the input arriving from the IM2COL unit requires less memory bandwidth. We can achieve high PE utilization in the GEMM unit with less throughput in the IM2COL unit. Thus, our IM2COL unit can be built with fewer resources and memory bandwidth requirements. Second, the tall array allows our design to take advantage of sparsity in the output of the IM2COL unit. As the width of the tall array is smaller than its height, fewer columns from the IM2COL transformation enter the systolic array at any instant of time, which increases the opportunity for detecting and skipping entire rows of inputs with zeros before entering the systolic array. By using a tall-shaped array, we simplify our algorithm by skipping redundant computations involving zeros in the input feature map. Section 3.4.3 describes our techniques for handling sparsity.

The GEMM unit employs an output-stationary dataflow, where a processing element

(PE) computes the final result by accumulating partial products for a particular output element. This output-stationary dataflow ensures maximum reuse of the output data. In addition, by using a tall array, SPOTS can achieve high data reuse for the result of the IM2COL transformation (i.e., feature map input). In addition, the output-stationary dataflow eliminates the need for separate multiplication and accumulation units. As a result, multiple levels of multiplication and addition are eliminated, as are routing logics across the two units (Section 3.2.3). Figure 3.8(a) shows the weight matrix from the filter and the output of the IM2COL transformation that forms the input to the GEMM unit. The values of the filter matrix enter the GEMM unit's systolic array from left-to-right. While the result of the IM2COL unit enters the systolic array from top-to-bottom. Figure 3.8(c) shows the various steps and partial results computed in the GEMM unit. Our design is parameterizable with M rows and N columns in the systolic array.

Our specific prototype used 128 rows of PEs and 4 columns. These numbers were chosen based on the characteristics of CNN layers. Each row of the systolic array can have multiple rows of the filter matrix assigned to it, depending upon the scheduling mode. There are fewer than 512 rows of filter matrices in the majority of layers in state-of-the-art CNNs.

In each PE, there is a single multiply-accumulate (MAC) unit that uses two 16-bit fixed-point inputs and accumulates the results in a 24-bit register. In our design, we use four K registers per PE to handle multiple rows of the filter matrix (e.g., in our design, $K = 4$). Each PE has three FIFOs. Two FIFOs, one for each arriving input. The other FIFO serves as the work queue for the MAC unit. The coordinates of the elements of the two input matrices should match before multiplying them in GEMM. As long as the inputs are sent to the PEs in the right order, we do not need additional logic to perform index matching inside a PE. Additionally, our output-stationary dataflow ensures that every partial product produced in a PE belongs to the same output element. Next, we describe how to support sparsities in both inputs without using any index matching units inside the PEs.

3.4.3 Handling Sparsity in CNNs

Most CNNs have sparsity in both filters and the input feature map. Figure 5.1 quantifies the amount of sparsity (percentage of zeros when compared to the total number of elements) for the commonly used CNNs. We use structured sparsity learning (SSL) [135] for our pruning method and optimize it to suit our hardware design better (Section 5.1). To store pruned weights, we propose a new sparse format. Our sparse format delivers high bandwidth access to the filters necessary to keep the PEs in a tall systolic array active. Our sparsity-aware design identifies and skips the zeros on-the-fly and at block granularity. In our sparsity-aware design, we skip zero blocks instead of individual zeros. That simplifies our sparsity-aware design. In addition, SPOTS skips zero blocks outside the PEs and in the input controller. Thus, we avoid using expensive hardware units for index matching or introducing any redundant multiplications (Section 3.2.3).

Our sparse format for weights. Once the weights for the filters are learned during the training phase, we divide them into blocks. The block size is equal to the group size used for pruning, which is a design parameter. Logically, the filter matrix will be a 2-D matrix of blocks when viewed in the dense representation. To minimize the memory footprint for storing the filters during the inference, we convert them into a sparse representation that is aware of the number of banks in the SRAM. To store the pruned weights compactly, we use three arrays. Figure 3.10a shows our custom sparse format. We store all non-zero blocks separately in one array (Array A) that is distributed in multiple banks based on the row index of the block (i.e., vertical position in the filter matrix). We use two bitmap arrays M1 and M2 to store the metadata. The bitmap array M1 indicates whether a column in the filter matrix has any non-zero values. A zero in the bitmap array M1 indicates an empty column. The bitmap array M2 maintains whether a block in a non-zero column is non-zero. A zero in M2 indicates the corresponding block is zero (i.e., as a block is a collection of values, it implies that all values in the block are zeros). Three arrays of our sparse format (i.e., A, M1, and M2) are distributed across different banks of the SRAM so that the input

controller for the GEMM unit can access them in parallel.

Figure 3.9 compares the memory footprint of our format with some of the most commonly used sparse formats in prior work. Unlike sparse formats like run-length encoding (RLC), CSR and DCSR, our format does not require additional storage to keep the count of the non-zeros (*e.g.*, RLC) or *data pointers* (*e.g.*, row pointer in CSR). Thus, the metadata size for our proposed sparse format is independent of the sparsity of filters, and it only depends on the total number of blocks in the weight matrix. Another important benefit of our sparse format over index-based sparse formats such as CSR is that it allows the non-zeros of a column to be stored in multiple banks. The banks can be processed independently and in parallel. CSC, CSR, and DCSR do not have this feature since they only keep the beginning of a column (CSC) or a row (CSR and DCSR). As shown in Figure 3.9, our sparse format outperforms other sparse formats for various density ratios. Compared to other bit-mask sparse formats like the one used in SparTen or Sparse-PE, our sparse format needs nearly $8\times$ less metadata by using the mask bits in a more coarse-grained fashion (*i.e.*, block level).

In summary, using a structured pruning method together with a proper sparse format enables SPOTS to gain a meaningful advantage over other sparse formats in storing the pruned weights. This can directly translate into energy consumption savings since the memory accesses (including both SRAM and DRAM accesses) are the main contributors to the overall energy consumption, as previous studies show [21].

Skipping zeros in the feature map and weights. The *compress* component before the GEMM unit in our accelerator (Figure 3.6a) identifies a block of zeros in the result of the IM2COL transformation. For each block coming out of the IM2COL unit, a bitmap is created. If all elements in a block in the output of the IM2COL unit are zeros, the bit is set to zero for that block; otherwise, the bit is set to one. Subsequently, the input controller of the GEMM unit uses this bitmap and M1 level bitmaps for the weights (Figure 3.10a) to skip blocks of the input feature map and weights on-the-fly when they are all zeros.

One unique feature of our approach is that we skip MAC operations involving zeros

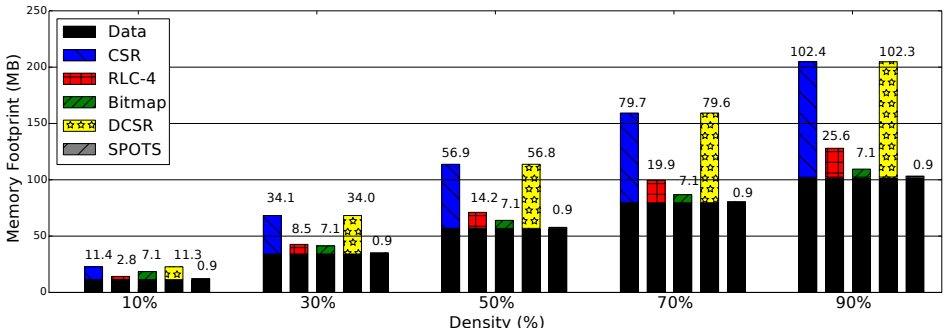
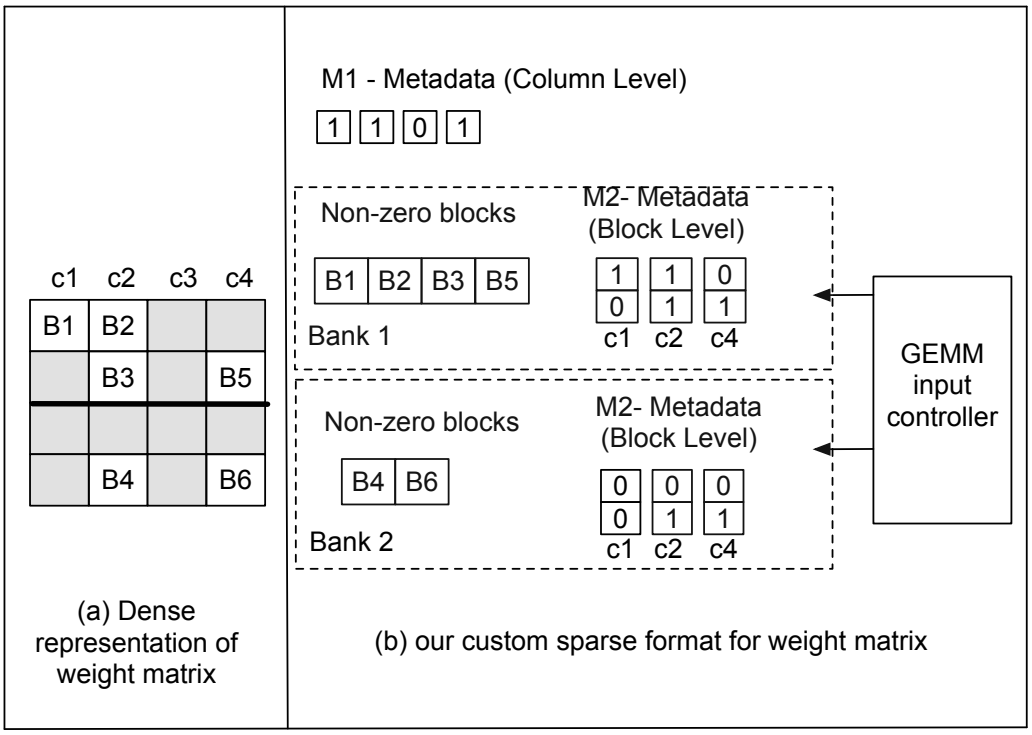


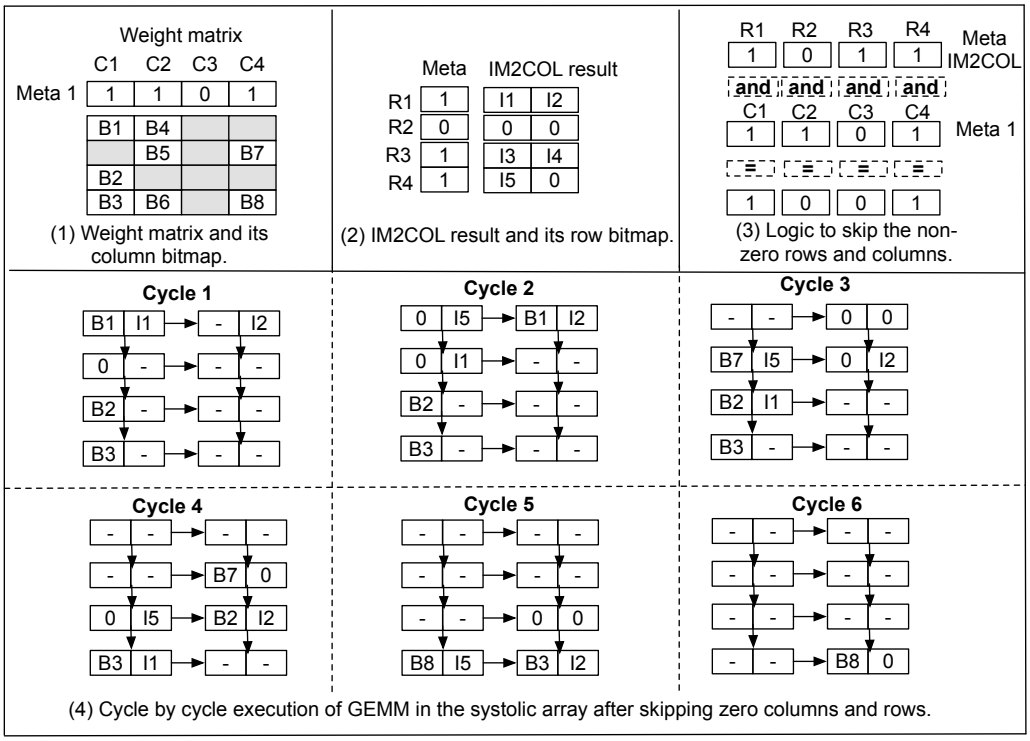
Figure 3.9: Comparing our custom sparse format with other state-of-the-art sparse formats. We used a matrix with 1632 rows and 36548 columns. We assume the values are 2 bytes. The last bar shows our sparse format. Our sparse format is independent of the density of the zeros in the matrix, and the size of metadata is less than 1 MB across all the density ratios. We compare our sparse format with CSR, RLC-4, Bitmap, and DCSR in the following order. The size of the sparse metadata is shown at the top of each bar in Megabytes.

outside the PEs and in the input controller. These have two advantages. First, we avoid the unnecessary data traffic to stream the rows of feature maps and columns of filters to PEs when they are zeros. Second, by detecting and skipping zeros centrally (inside the input controller), the PEs are relieved of storing and processing any metadata, thus reducing the amount of area and power consumed. Besides, our approach does not require any expensive hardware units inside every PE to detect and match the non-zero pairs, unlike some prior work (Section 3.3.3). Figure 3.10b illustrates how the zero columns in the weight matrix and the zero rows in the output of the IM2COL unit are skipped. In addition to the zero blocks that we skip in the control unit, some PEs may still receive zero blocks (the gray blocks in C1, C2, and C4 columns in 3.10b). This happens when a column of the weight matrix is partially zero. In those cases, the input controller sends one bit to the PE to indicate a zero block. As a result, PEs ignore blocks with all zeros and MAC units are gated to reduce energy consumption.

Finally, we highlight the role of the tall systolic array in our on-the-fly detection of the non-zero blocks in the feature map. The number of elements entering a tall systolic array is limited to blocks with a small number of elements (e.g., blocks consist of 4 elements in our prototype). Small block sizes increase the possibility of detecting blocks that include only



(a) Our custom sparse format to store filters



(b) Skip rows and columns with all zeros

Figure 3.10: (a) Our custom sparse format to store filters. (b) Illustration of how our design skips rows and columns with all zeros. (1) Weight matrix with the metadata about columns with all zeros. (2) The IM2COL result with the metadata about rows with all zeros. (3) If a row or a column is all zeros, all such rows and columns can be skipped (i.e., *and* operation of the row and column metadata). (4) GEMM computation when rows and columns are skipped. For example, the first element of column C4 will be fetched by the first PE in cycle 2 (skipping columns C2 and C3).

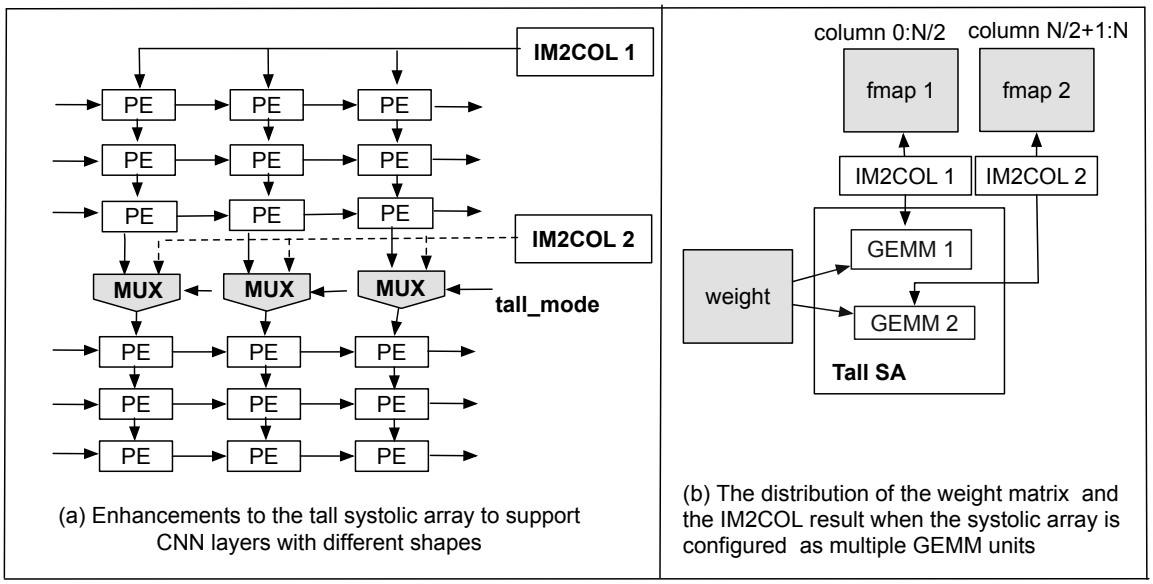


Figure 3.11: (a) Enhancements to reorganize the tall systolic array (SA) as multiple GEMM units. (b) Illustration of how inputs are distributed in the configuration with multiple GEMM units.

zero elements and are easier to skip. The $\star$ marker in Figure 5.1 indicates the percentage of zeros skipped on-the-fly with this technique in the output of the IM2COL transformation.

3.4.4 Handling Various CNN Layers/Shapes

CNNs have multiple layers that can vary in size and shape. When PEs have a fixed configuration, they can be underutilized for some layers' shapes and sizes. Each filter forms a row of the weight matrix that is assigned to a distinct row of the systolic array. When the GEMM unit is configured as a tall systolic array, and the number of filters is relatively smaller than the systolic array's height (e.g., 128), some PEs will remain unused.

With GEMM's dynamic reconfigurability, we are able to support CNN layers with a variety of attributes (Figure 3.11). Specifically, the PEs in the GEMM unit can be configured either as one tall array or multiple small arrays. Both configurations have the same number of columns. This enhancement allows our design to be more **adaptive** to different layer shapes and thus maintains high PE utilization under different conditions. Figure 3.11(a) demonstrates how a tall array can be used as two smaller arrays using the multiplexers.

Hence, the PEs now can receive the input either from the PEs above (i.e., it forms a tall array) or can get the input from a different IM2COL unit. Depending on the mode register and the features of a layer, these multiplexers can be configured dynamically. The weight matrix is broadcast to all small systolic arrays when the GEMM unit is configured as smaller systolic arrays. Each small GEMM unit receives the feature map input from their assigned IM2COL units. The two GEMM units compute two independent groups of columns of the final result matrix (i.e., GEMM 1 computes the result columns from 0 to N/2, GEMM computes the columns from N/2+1 to N). In our prototype, we have four IM2COL units. There is one main IM2COL and three smaller IM2COL units to support the two configurations. The main IM2COL unit is used for the tall array configuration. The other configuration uses all four IM2COL units. This dynamic reorganization of the GEMM unit's systolic array coupled with the multiple IM2COL units enables our hardware to maintain high PE utilization for various CNN layers with different shapes.

Supporting fully connected layers. Most CNNs have one or more *fully connected* layers at the end of the network. The fully connected layers use the matrix weights learned during the training and the output feature map provided by the final convolutional layer that is flattened to a vector. The computation of the fully connected layer is equivalent to matrix-vector multiplication for a batch size of 1. We can structure it as a matrix-matrix multiplication by increasing the batch size. As we use a tall array, the batch sizes need not be large to utilize the whole array of PEs fully (*e.g.*, can be as small as 4).

Supporting pooling layers. A pooling layer summarizes the features generated by a convolution layer. The two most common types of pooling layers are max pooling and average pooling. Among them, max pooling, which picks the maximum element from a feature covered by the filter, is more common. Similar to convolution layers, the pooling layer has two parameters, filter size, and stride size. We support the pooling layer by adding the pooling operation (e.g., MAX) to the output of the patch units (PUs) in the IM2COL unit.

3.4.5 Strategies to Improve Load Balance in SPOTS

Load imbalance happens in sparse CNNs due to the uneven distribution of the non-zeros in weight and feature map inputs. An accelerator's choice of dataflow and data reuse strategy determines the source of load imbalance. Generally, accelerators adopt either an input-stationary or an output-stationary dataflow. Subsequently, an input-stationary dataflow can be weight-stationary or feature map-stationary. The input-stationary dataflow keeps one input stationary in the PEs, while the other input is broadcast to each PE to ensure data reuse. When there is an uneven distribution of non-zeros in the inputs, some PEs may receive fewer inputs, forcing them to remain idle until the other PEs process their inputs before they all can receive new inputs.

SPOTS adopts an output-stationary dataflow with a tall systolic array (Section 3.4.2). In a tall systolic array, the feature map values are passed through as many PEs as possible to ensure maximum data reuse. We skip the zeros in the feature map input inside the input controller before entering the systolic array, as described in Section 3.4.3. Thus, the non-zeros are skipped for *all* PEs (not just for an individual PE) in the systolic array. SPOTS's early zero detection method avoids load imbalance caused by the uneven distribution of non-zeros in the feature map. In a similar way, SPOTS detects and skips the zeros in the weights outside the PE when they span the whole filters (*i.e.*, an entire column of the weight matrix).

As a result of partially zero columns in the weight matrix (i.e., some blocks are zeros, others are non-zeros), some PEs may receive zero blocks, while others receive non-zero blocks. This can lead to an imbalance in work between PEs. The load imbalance among the PEs can be quantified using the metric proposed by [33] where the load imbalance is quantified as follows:

$$imbalance\_percentage = \frac{maximum\_work - average\_work}{maximum\_work} \times \frac{n}{n-1} \tag{3.1}$$

The imbalance percentage corresponds to the percentage of time the PEs with less work are not engaged in useful work and are waiting for the PE with the maximum work. A perfectly balanced work distribution results in 0 imbalance percentage. Thus, a lower imbalance percentage results in fewer idle cycles for the PEs.

One way to improve the load balance in the PEs is to rearrange (shuffle) the non-zero blocks in the weights offline to make the distribution of the non-zero blocks more balanced. However, this reshuffling can change the position of the output channels and thus requires an additional step to reorder the output before the next layer uses them [42, 69]. In SPOTS, we did not add any additional load balancing hardware or software units to avoid creating additional complexity. In Section 5 we present the average imbalance percentage for all four CNN architectures with SPOTS.

3.5 Related work

There is a substantial body of literature on using custom hardware accelerators to improve neural network performance and energy efficiency [8, 19, 21, 30, 36, 40, 48, 55, 83, 98, 102, 103, 145]. Table 3.1 provides a qualitative comparison of SPOTS with closely related work.SPOTS supports various operations in CNNs, is adaptive to layers of different shapes with high PE utilization, and effectively supports sparsity both in the feature map and in the weights.

Support for sparse inputs. Prior work has improved energy efficiency by supporting sparse inputs during inference. Cnvlutin [8] exploits sparsity in the input feature map to skip multiplication operations and avoid data movement with zero elements. CambriconX [145] supports sparsity in the weights. Similar to our work, SCNN [98] and CambriconS [149] support sparsity in both the feature map and the weights to improve energy efficiency and performance. Previous work has used data gating techniques to reduce the power consumption when the operands are zeros [21, 103]. The zero value clock gating technique reduces power consumption without reducing the number of effective operations.

Similar to SPOTS, previous hardware designs have developed techniques to skip zeros and minimize the amount of data transferred [30, 48, 90, 98]. Prior work use both random and structured pruning techniques. EIE [48] uses a random pruning method coupled with a fine-grained sparse CSR-encoded to store the pruned weights. ESE [47] extends this approach to Long short-term memory (LSTM) neural networks. Similarly, MASR [45] and Doping [126] exploit an unstructured sparsity for RNNs and LSRM, but instead of a CSR format, they use a bitmask encoding. Similar to SPOTS, prior studies propose various structured pruning methods and hardware methods to take advantage of the sparsity while avoiding the complexity of random pruning methods. Kang [63] proposed a scheme that prunes the weights so that the number of weights for each weight group remains the same. This addresses some of the inefficiencies of random pruning, such as the internal buffer misalignments and load imbalances. A recent work [83] uses the same approach but instead applies it to both weight and feature map input.

Support for various layers in CNNs. Many hardware designs are tailored for a single type of computation and do not support all types of layers in CNNs, such as pooling layers [69]. EIE [48] is intended for the fully connected layers in CNNs. The input feature map and filters are stored in a compressed format, and only non-zero operands are passed to the multiplier. In contrast, SCNN [98] and Eyeriss [20, 21] primarily focus on the convolution layers. This means they can underperform for the fully-connected layers. SCNN can achieve 25% of peak throughput when performing the fully connected CNN layers. Eyeriss provides significant energy gains only for batches larger than 16. In contrast, SPOTS supports all the common layers that exist in CNNs.

Systolic array designs for CNNs. Tensor Processing Unit (TPU) [60] is an ASIC that has matrix multiplication as its core computation block to accelerate CNNs. TPU requires the use of the host CPU to perform some data reorganization and does not support sparse inputs. Recent work [51, 69] uses a preprocessing step (i.e., column combining) to pack a sparse CNN into a denser form before passing the inputs to a systolic array for GEMM.

Table 3.1: Qualitative comparison of SPOTS with prior work.

Accelerator	Supports sparsity				Supports pruned network	Adaptive to different layer shapes
	Feature map	Weight	Gate zero	Skip zero		
Eyeriss [21]	✓	✗	✓	✗	✗	✗
Cnvlutin [8]	✓	✗	✓	✓	✗	✗
CambriconS [149]	✓	✓	✗	✓	✓(structured)	✗
SCNN [98]	✓	✓	✓	✓	✓(random)	✗
CMSA [139]	✗	✗	✗	✗	✗	✓
Column combining [69]	✗	✓	✗	✓	✓(structured)	✗
SIGMA [101]	✓	✓	✗	✓	✓(random)	✓
Sparse-PE [102]	✓	✓	✗	✓	✓(random)	✗
SPOTS (this work)	✓	✓	✓	✓	**✓structured**	✓

It is unclear how to prepare input feature maps for matrix multiplication. The method will not be useful if the input feature map is highly sparse. Our group-wise pruning provides higher accuracy than the column combining method. Simultaneous multithreaded systolic array (SMT-SA) [112] addresses the underutilization and load imbalance introduced by random pruning of the weights in a CNN. In Similar fashion to SPOTS, recent work [82] utilizes a structured pruning accompanied by a novel data format called density-bound block (DBB) better to map the sparse inputs to the systolic architecture. Gemmini [40] is another accelerator that uses a GEMM to accelerate CNNs. Both software and hardware IM2COL units are explored in their work. Similar to our work, their results demonstrate that using a hardware IM2COL can significantly improve performance. As opposed to SPOTS, Gemmini does not take sparsity into account. Furthermore, PEs are rigidly organized in their design, resulting in underutilization of PEs for certain layer shapes. Recent work [150] proposes a memory-efficient hardware for the IM2COL. The main insight is to layout the feature map elements in the SRAM in a *Channel-First* manner. In this way, the feature map inputs are sent one column at a time to the GEMM unit. We use multiple PUs connected by a ring network instead of their single PU. This allows us to generate multiple columns of inputs at the same time. Their design stores the feature map elements in one large SRAM. Instead, we use multiple smaller SRAMs to store the feature maps to save energy.

Flexible interconnects. The flexibility of the interconnect between PEs can help support different filter sizes [71, 101]. Maeri [71] enables tree-based reconfigurable interconnect networks to facilitate DNN accelerators dataflow mapping. The downside of MAERI

is that it cannot handle sparsity in input feature maps. Similarly, FlexFlow [86] develops a flexible dataflow architecture that uses different types of parallelism along with different CNN workloads. As opposed to them, SPOTS uses a regular interconnect network between the PEs. Another recent work SIGMA [101] proposes a flexible non-blocking interconnect to achieve high compute utilization across layers of varying shapes. The SIGMA architecture is primarily designed to handle high-precision inputs during the training phase. Moreover, they only focus on the GEMM and do not study the IM2COL transformation and do not support other types of layers CNN. Recent work [139] design a configurable multi-directional systolic array (CMSA) that improves the PE utilization for small-scale convolution or depthwise convolution. Yet, their design focuses solely on increasing PE utilization and does not take into account other aspects, such as sparse input and IM2COL design.

Load balance in sparse CNN accelerators. Load balancing is not supported by many hardware accelerators for sparse CNNs [90, 98, 102]. GoSPA [30] employs a passive strategy to deal with load imbalance problems. To maintain high multiplier utilization in the presence of load imbalance, it employs a two-stage buffering technique. Using large buffers can negatively impact their design's area and power consumption. By contrast, SparTen [42] and Column Combining [69] use a systematic approach to deal with load imbalance in their designs. SparTen proposes a greedy balancing method with two variants: a pure software approach and a software-hardware hybrid. SparTen also balances load at two granularities (e.g., at the filter and chunk levels). The load balancing at a finer grain (i.e., chunk level) requires a hardware multi-stage permutation network to *unshuffle* the partial sum of each chunk to the appropriate output sum. Column Combining [69] suggests a new method for packing sparse CNNs into a denser format for efficient execution using systolic arrays. They combine multiple sparse columns of a convolutional filter matrix into a single dense matrix. Like SparTen, they introduce extra hardware to permute the rows.

Bitwise sparsity. Some prior studies identify and skip the zero computations at the bit

level [7, 29, 108]. Pragmatic [7] adopts the bit-sparsity for the weights. Tactical [29] targets bit-sparsity in the feature map input instead. By doing that, they uncover more ineffectual work. Laconic [108] outperform both designs by applying the bit-sparsity for both weight and feature map inputs.

3.6 Summary

This chapter introduces SPOTS, a hardware accelerator for sparse CNNs using matrix multiplication with the IM2COL transformation. The hardware IM2COL unit reads the input feature map only once, reuses the data, and executes in parallel with a tall systolic array for the GEMM unit. We add flexibility to the systolic array that allows it to achieve high PE utilization for CNN layers of varying sizes and shapes. SPOTS supports sparsity both in the input feature map and the filters. Similar to our design in Chapter 2, we use the software to preprocess the sparse data and organize them into a hardware-friendly format. This allows the hardware to access the input with high bandwidth. Unlike the design for SpMV and SpGEMM, the software preprocessing step is not overlapped with the hardware execution. Instead, the software preprocessing happens prior to the hardware execution and its cost is amortized over many executions of the hardware. We evaluate the performance and energy efficiency of our accelerator and compare it with other recent hardware accelerators for CNNs in Chapter 5.

CHAPTER 4
AN FPGA ACCELERATOR FOR SPARSE CONVOLUTIONAL NEURAL NETWORKS

In this chapter, we present our FPGA implementation of SPOTS. In Chapter 3, we presented an ASIC design of SPOTS for accelerating the inference of sparse convolution neural networks. In SPOTS, we formulate the convolution layers as general matrix-matrix multiplication (GEMM) with an Image to Column (IM2COL) transformation and offload both computations to the hardware. The process of designing and manufacturing ASICs can take a long time and can be expensive. FPGAs are an alternative to designing custom hardware. FPGA reconfigurable substrates reduce non-recurring engineering (NRE) costs and can be reprogrammed for different applications. This chapter presents our FPGA design for the inference of sparse convolutional neural networks. We use the same convolution formulation as Chapter 3. By offloading both IM2COL and GEMM computation to the FPGA, we reduce CPU and FPGA data traffic. The ASIC and FPGA are different in some respects. A main difference between FPGAs and ASICs is that FPGAs operate at a lower frequency due to their reconfigurability. Hence, we revised some of the components of our design in Chapter 3 to better map onto FPGA.

Unlike most prior FPGA designs for CNNs that support only the sparsity in the weight input, our FPGA design exploits sparsity in both weights and feature maps. We avoid complex hardware units for skipping zeroes by using a tall-thin structure of PEs along with a hardware-friendly pruning method. Our sparsity-aware design suits FPGA architecture. In addition, we enhance our design to have good performance for CNNs with different layer shapes. Further, our design can scale up or down depending on the available resources of a target FPGA. Finally, we demonstrate how different components of our design can be realized using High-Level Synthesis (HLS) tools for FPGAs.

4.1 Background on High Level Synthesis for FPGAs

We provide background on FPGA programming using the High Level Synthesis (HLS) tool. While our examples use the Xilinx HLS syntax, these concepts also apply to other HLS tools.

4.1.1 Building FPGA Designs with High Level Synthesis

Register transfer level (RTL) design using hardware description languages such as Verilog, VHDL, and Bluespec is the gold-standard programming model for FPGAs. The process of describing a design in RTL is often compared to writing assembly code for a CPU [93]. Over the past few decades, numerous attempts have been made to simplify the process of building applications for FPGAs. One notable example is High Level Synthesis (HLS). The HLS toolkit provides an API for FPGA designers to convert designs described in high-level programming languages, such as C/C++, Scala, or Haskell, into RTL code. There are several well-known HLS tools such as Vivado HLS [95], Synopsys Synphony C Compiler, and LegUp [17].

While HLS tools relieve the designer from some of the difficulties of building custom hardware on FPGAs, building an efficient design still requires a designer with extensive application domain knowledge and familiarity with the hardware architecture. Next, we describe some important optimizations that can improve the performance and efficiency of the FPGA design while using HLS tools.

4.1.2 HLS Optimizations

HLS tools use a C/C++ front end and a set of transformation heuristics to map software constructs onto hardware elements along with a back end that generates RTL code [17, 24]. To satisfy resource, layout, and timing requirements, a constraint solver is typically deployed [25]. To guide the transformation, programmers can add *#pragma* hints. HLS

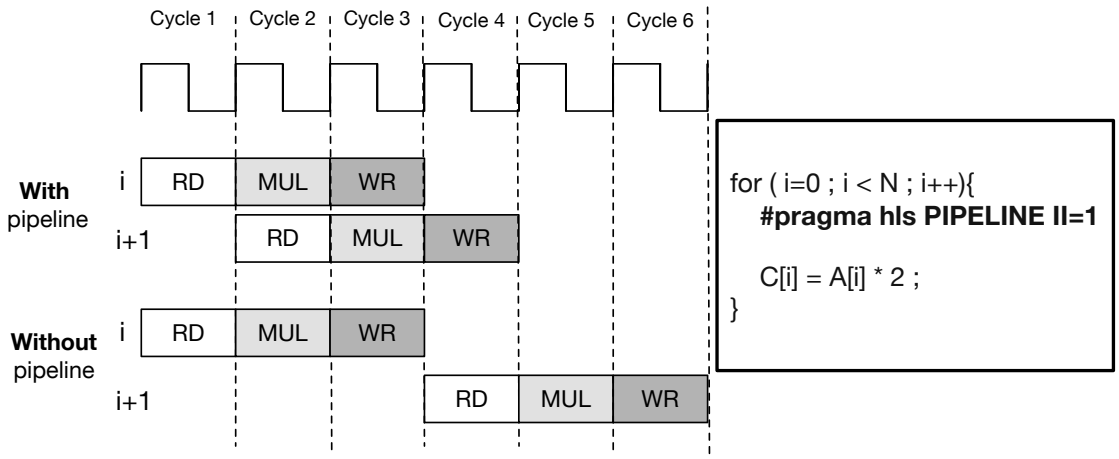


Figure 4.1: Example of a loop pipelining. A designer can indicate the desired initiation interval by setting II. The compiler pipelines the loop iteration automatically if there is no data dependency between iterations. In this example, the initiation interval is set as 1. This means the next iteration starts as soon as the next cycle.

tools make a heuristic effort to translate any valid C/C++ program to RTL. In this section, we present some of the most important pragmas that programmers can use to improve their design performance in HLS.

Exploiting Parallelism. FPGAs enable designers to extract parallelism at a finer granularity to improve performance. Parallelism can be defined at various levels in HLS. It can be the operations within a loop or the parallelism between multiple functions. Programmers can use directives (via pragmas) to express parallelism in their code.

Loop pipelining. Loop pipelining is used to define parallelism at the instruction level. Figure 4.1 shows an example of a simple loop that performs simple arithmetic operations on two vectors (A and C). The loop includes a read operation to read one element of input A, a multiplication operation, and a write operation to write the result to the output array (C). Figure 4.1 shows the loop execution cycles *with* and *without* the pipelining. A loop **without** the *pipeline pragma* processes the three operations sequentially and waits for all the operations in the previous iteration to finish before starting the new iteration. When a pipeline pragma is used, the next iteration is started immediately after the module can

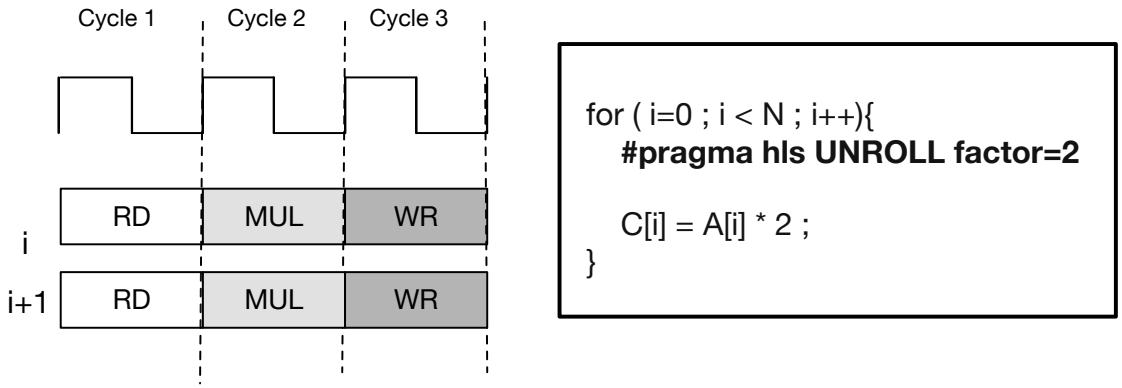


Figure 4.2: Example of unrolling a loop. A designer can indicate the desired unrolling factor by setting unroll factor. In this example, the unrolling factor is 2, meaning two copies of the modules will execute the loop in parallel.

accept new input. The initiation interval (II) defines how fast the next iteration can begin. II is set at the cycle level. In general, it is desirable to have loops with II=1. In some cases, the tool may not be able to reach the designer's desired initiation interval. For example, loop carry dependencies can prevent loop pipelining. When this occurs, the tool selects the minimum possible II from that loop. Several other pragmas can be used to provide additional information to the tool to overcome loop-carry dependencies, but we do not explain them here.

Loop unrolling. Unrolling is another type of parallelism that can be defined within a loop. Unrolling makes multiple copies of the operational module that can then execute in parallel. A user can set the unrolling factor. Figure 4.2 shows the earlier example (Figure 4.1) this time with an unrolling pragma. In this example, the unrolling directive instructs the HLS tool to create 2 copies of the multipliers. A loop can be unrolled partially or completely. Loop unrolling represents an area/performance trade-off. Unlike loop pipeline, unrolling can only be applied when loop iterations are known at compile time. Similar to loop pipeline, the compiler may fail to achieve the designer requested unrolling factor. Insufficient resources are one of the most common reasons for this. In our example, if there are not enough memory ports to read input for multiple copies of the loop body, then

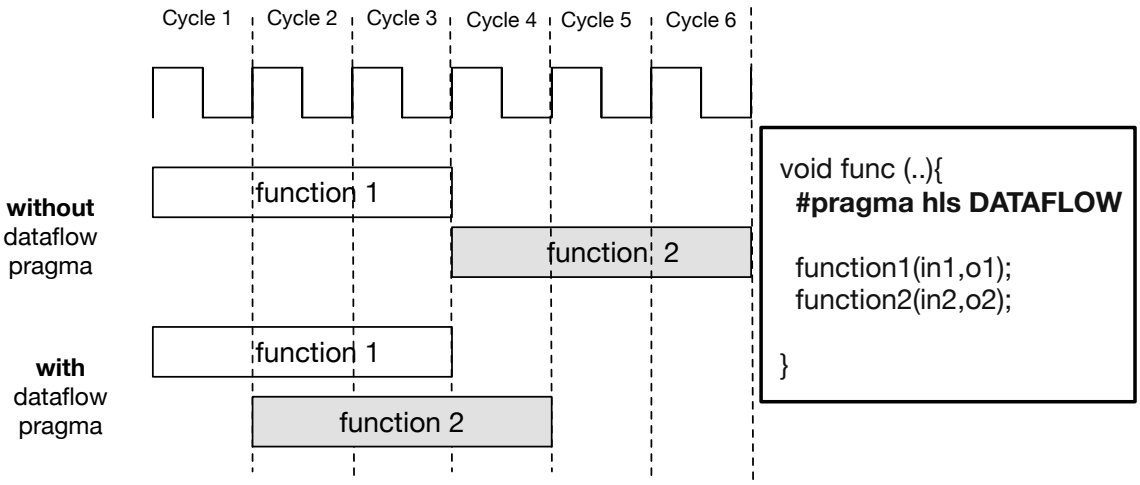


Figure 4.3: Example of using the dataflow pragma. With a dataflow pragma, function 2 can start right after function 1. This saves 2 cycles compared to the design without a dataflow pragma.

the compiler will not be able to unroll the design as requested. We will discuss how the memory units can be partitioned to meet the requirements for the hardware modules later.

Task-level Parallelism. The parallelism can be defined at a higher level such as the entire task. *Dataflow pragma* is used to define parallelism at the function level. A Dataflow pragma pipelines the functions and schedules them to start their operation as soon as the inputs are ready. Figure 4.3 presents the task level pipelining. The overall latency to finish two functions is 6 cycles (three for function 1 and three for function 2). When they are pipelined, the second function can start its process after the first cycle. Thus, the overall execution time is reduced to 4 cycles with dataflow. In pipelining the tasks, the initiation interval is determined by the task with the highest latency.

Memory configuration. The improper memory partitioning is one of the main reasons why the design cannot achieve the desired parallelism (II=1 for the initiation interval for loop pipelines or the maximum unrolling factor). It is the designer's responsibility to choose optimal memory partitioning for the design. In addition, FPGAs often offer various on-chip memory resources such as block-RAMs (BRAMs), LUT RAM, and Ultra RAM (URAM). Utilizing all the available resources can help to improve the overall performance

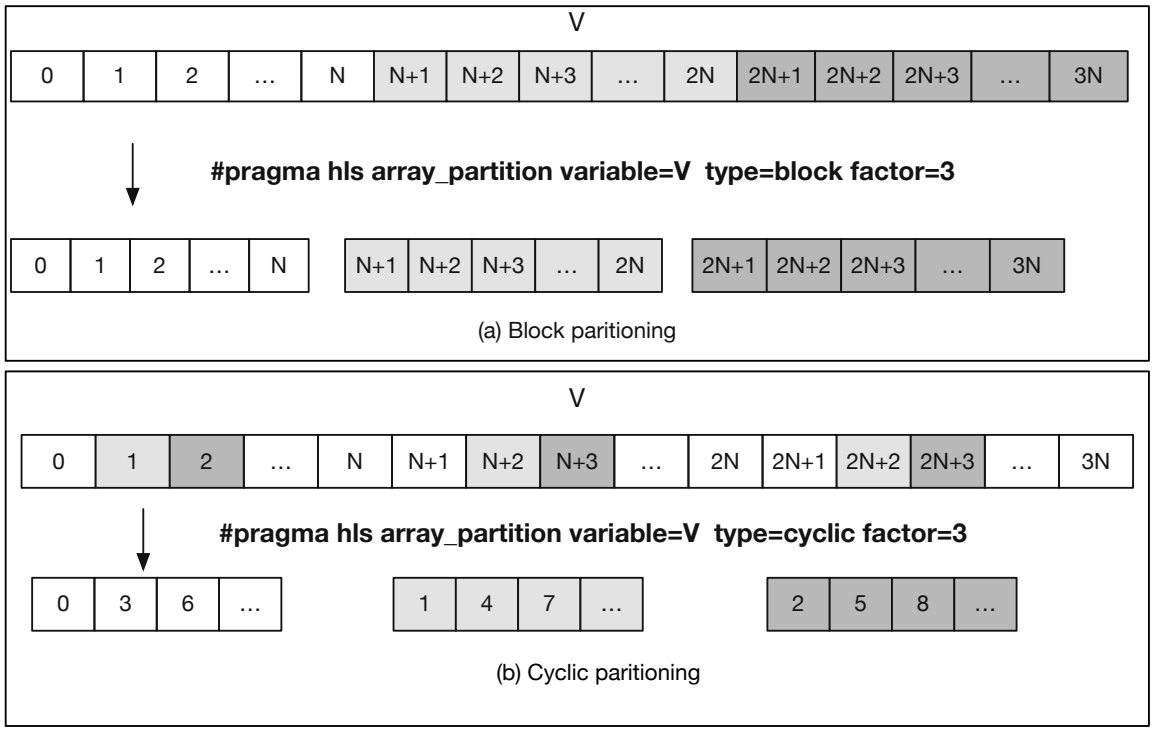


Figure 4.4: Example of partitioning an array that splits a large array (V) into multiple smaller arrays. (a) shows a *block partitioning* and (b) shows a *cyclic partitioning* of array V.

of the design.

Memory partitioning. Memory partitioning divides a single array of data into multiple arrays and assigns each array to a different memory resource. Each memory module can be accessed independently. Figure 4.4(a) demonstrates a *block paritioning* method where each smaller array is created from consecutive blocks of the original array. Another way to partition an array is *cyclic partitioning* that creates smaller arrays by interleaving elements from the original array (Figure 4.4(b)).

Specifying the memory resources. The designer must be aware of the resources available on the target FPGA. HLS tools often report the resource utilization of a design. Exhausting the resources such as the on-chip memory can result in a timing failure or a low design frequency. HLS tools allow the designer to request a specific resource through pragmas.

The compiler silently ignores the designer's request if the target FPGA does not support the requested resource.

4.2 The Architecture of Our FPGA Accelerator for Sparse CNNs

In this section, we present the architecture of our FPGA accelerator for sparse CNNs inference. Similar to our design in Chapter 3, we formulate the convolution operation as a GEMM using an IM2COL transformation. Our design goals are three-fold: (1) to pipeline the IM2COL operation with the GEMM computation to reduce data traffic between the CPU and FPGA and improve the latency, (2) to exploit the sparsity in the weights and the feature maps, and (3) to design a flexible system that can adapt to different CNN layer shapes and sizes while scaling up or down based on FPGA resources.

First, we present the GEMM and IM2COL units. Then, we describe our sparsity-aware design and how it is suitable for FPGAs. Finally, we explore how our design can adjust to different layer shapes while scaling to FPGAs with varying resources.

4.2.1 The IM2COL Unit

The IM2COL transformation creates a 2-D matrix from the 3-D input feature map, which reduces a convolution operation to a general matrix-matrix multiplication (see Section 3.2.2). Section 3.4.1 presents the IM2COL unit design targeting ASICs. The proposed IM2COL unit is a collection of patch units (PUs) connected via a ring network (see Figure 3.6b). Each PU includes three local buffers. These local buffers are used to store the overlapped elements among patches. As a result, expensive DRAM accesses are minimized and throughput is increased. Since FPGAs have limited and fixed memory resources, we had no choice but to minimize the on-chip memory resources for our IM2COL unit. In the FPGA version of SPOTS, we reduce the memory requirement for the IM2COL unit by removing the three local buffers that exist in PUs in the ASIC design. Accordingly, we modify our patch generation algorithm based on these changes. When the local buffers are removed, some of

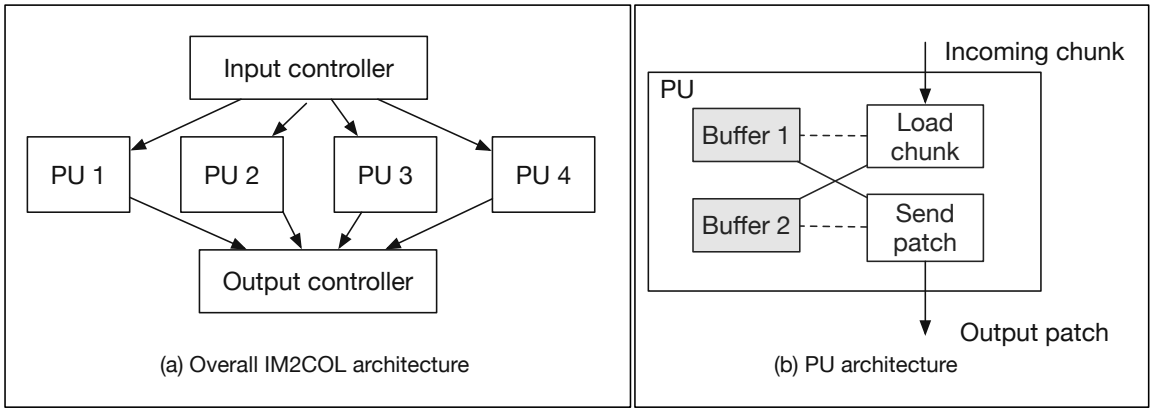


Figure 4.5: A high-level overview of IM2COL and patch units.

the elements are accessed multiple times from external memory. In addition, the IM2COL throughput is reduced compared to the design proposed in Section 3.4.1. As our FPGA design operates at a lower frequency than our ASIC design, the lower throughput of the IM2COL unit has a minimal impact on the GEMM unit performance.

Figure 4.5(a) shows the overall architecture of the revised IM2COL unit. The design consists of an input controller, multiple patch units (PUs), and an output controller. In the new design, the PUs are not connected and thus do not communicate. The input controller reads the elements from off-chip memory and distributes them among the PUs according to their *channel ID*. The PUs create the patches for different channels. Thus, we exploit parallelism at the channel level. The number of PUs is a design parameter. The input controller memory bandwidth and FPGA resources should be considered when determining the number of PUs. Finally, the output controller reorders the patches received from the PUs and sends them to the GEMM in the correct order. Next, we will describe each unit in detail.

Input controller unit. Figure 4.6 illustrates how the input controller reads the input feature map and distributes it among the PUs. Elements of the feature map are fetched in chunks of size $K \times W$, where K is the kernel size and W is the input width (Figure 4.6(b)). The input controller distributes the chunks among the PUs based on the channel IDs in a

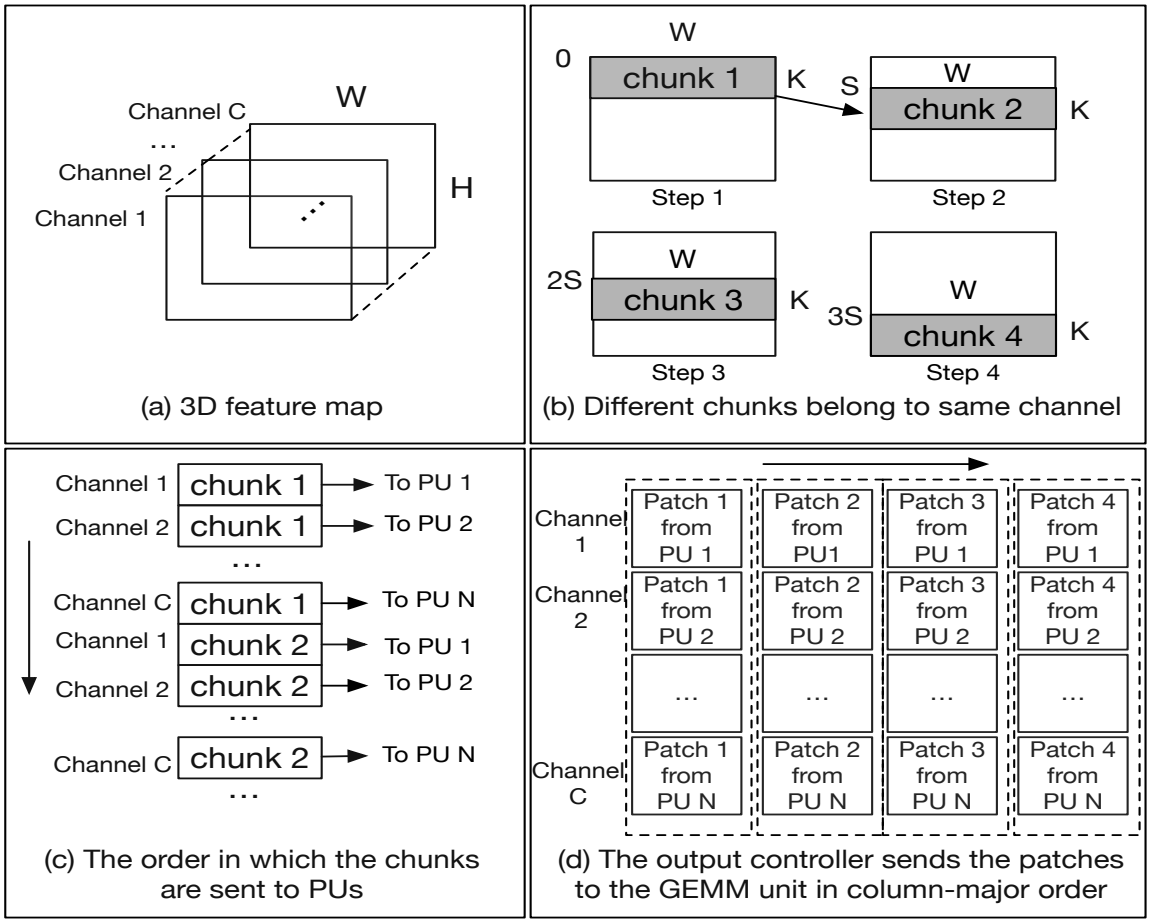


Figure 4.6: (a) The 3D format of the input feature map. (b) Illustrating the input chunks sent to each PU by the input controller (c) Illustrating the order in which the input controller sends different chunks to the PUs. (d) Illustrating the order in which the output controller forwards the patches received from the PUs to the GEMM unit.

round-robin fashion (Figure 4.6(c)). Each PU receives all chunks belonging to the same channel. Different chunks may overlap horizontally, depending on the kernel and stride sizes. The chunks overlap when the stride size is smaller than the kernel size. Unlike Section 3.4.1, we do not save the elements in local buffers so they can be reused to build later patches. Thus, the input controller may read some of the elements from external memory multiple times.

Patch unit (PU). Figure 4.5(b) shows the different components of a PU unit. In each round, the PUs receive the chunks for their assigned channels. They generate the patches

based on the order the GEMM unit processes the patches (column-major order). Listing 4.2 shows the nested loops to build patches in the PU unit. We use the double buffering technique to overlap, fetching the chunk into local memory and building patches. The two buffers (buffer 1 and buffer 2 in Figure 4.5(b)) need to be alternated between the module that fetches the chunks and stores them locally and the module that generates the patches and sends them out. Listing 4.1 demonstrates how the double buffering technique can be realized with HLS. The *load_flag* and *send_flag* indicate whether the *load_chunk* and *send_patch* units should be active or not. In the first iteration, the *load_flag* is set, and in the last iteration, the *load_flag* is unset. The *send_flag* is set after the first iteration and remains active for the rest of the iterations. Besides, in the even and odd iterations the *load_chunk* and *send_patch* modules access a different set of buffers. In the even iterations, *load_chunk* writes the chunks into the *local_buffer_1* and *send_patch* reads from *local_buffer_2*. Finally, lines 4 and 6 of Listing 4.1 also demonstrate how we utilize URAM memory instead of the default BRAM memory for the two local buffers.

As opposed to the PUs in Section 3.4.1, the revised design has only two sets of local buffers (due to the double buffering) to store the data chunks received from the input controller. In the revised design, the total on-chip memory usage for all the PUs is $C \times K \times W$, where C, K, W are the channels, the kernel size, and the input width, respectively. The width and the channel size are inversely proportional to each other. The layers with fewer channels are wider than those with more channels.

Output controller unit. The output controller receives the patches from different PUs. As the channels are distributed among the PUs in a round-robin fashion, patches should be ordered according to their PU ID. Output controller sends the complete patches to the GEMM unit in a column-major order (see Figure 4.6(d)). To store incoming patches, the output controller has a separate buffer for each PU. These buffers prevent the PUs from stalling while the output controller sends patches to other PUs.

```
void PU (stride_size,num_patches,incoming_chunk,output_patch_stream)
{
  DATA_TYPE local_buffer_1 [BUFFER_SIZE] ;
  #pragma HLS RESOURCE variable=local_buffer_1 core=XPM_MEMORY uram
  DATA_TYPE local_buffer_2 [BUFFER_SIZE] ;
  #pragma HLS RESOURCE variable=local_buffer_2 core=XPM_MEMORY uram
  for (int k=0 ; k < num_patches+1 ; k++) {
    #pragma HLS PIPELINE II=1
    int load_flag = (k >= 0 && k < num_patches);
    int send_flag = (k > 0 && k <= num_patches);
    if (k%2==0) {
      load_chunk(load_flag,incoming_chunk,local_buffer_1,...) ;
      send_patch(send_flag,local_buffer_2,output_patch_stream,...);
    }
    else {
      load_chunk(load_flag,incoming_chunk,local_buffer_2,...) ;
      send_patch(send_flag,local_buffer_1,output_patch_stream,...);
    }
  }
}
```

Listing 4.1: Demonstrating the double-buffering technique used in the PU units to overlap loading the data chunk and generating the patches.

```
void send_patch (flag,PU_local_buffer,output_patch_stream,...)
{
  const unsigned ch_k =  kernel_size * width ;
  if (flag){
    for (unsigned cc=0 ; cc+kernel_size-1 < width ; cc+=stride_size){
      for (unsigned ch=0,chnl=0 ; ch < num_channel ; ch+=NUM_PU,chnl++){
        for (unsigned rk=0 ; rk < kernel_size ; rk++) {
          for (unsigned ck=0 ; ck < kernel_size; ck++) {
            #pragma HLS PIPELINE II=1
            unsigned off_r = (chnl * ch_k) + cc + rk*width ;
            output_patch_stream << PU_local_buffer[off_r + ck] ;
          }
        }
      }
    }
  }
}
```

Listing 4.2: The nested loops to generate the patches using the chunks received from the input controller.

4.2.2 The GEMM Unit

We use a tall systolic array for the GEMM unit similar to the design in Chapter 3. We explained the advantages of using a tall systolic array and how to compute matrix-matrix multiplication with an output-stationary formulation with the systolic arrays in Chapter 3. In this section, we discuss the changes we made to the design in Section 3.4.2 to make it more FPGA-friendly. The primary difference between the two designs is how the processing elements (PEs) are organized in the systolic array. We describe the PE organization in the revised GEMM unit next.

PE organization. Figure 4.7 shows a 2D and 1D grid of PEs in the systolic array of the GEMM unit. Prior research has shown the advantages of using a 2D structure for ASICs [60, 69]. However, we found a 1D array of PE better suits FPGAs for the following reasons.

- Collapsing the columns of the 2D systolic array into one can reduce the design resource utilization. While the design still has the same number of MAC units, the logic for managing the buffers and MAC units can be shared. The buffers can be partitioned properly to allow parallel access to the elements. Additionally, using a 1D array of PEs reduces the number of FIFOs that connect the PEs. This can be accomplished by using FIFOs with wider interfaces that can be used to communicate inputs and outputs between PEs. Additionally, a 1D array reduces the routing logic between PEs and is more suitable for FPGAs.

- To write back the result matrix, each PE sends its output to the next PE (the PE at the bottom) after forwarding the output received from the previous PEs. We need N different output controller units to drain the output received from the N PEs in design in Figure 4.7(a), where N is the width of the systolic array. Using a 1D array of PEs, we only need one output controller unit to write the output to the external memory. Each external memory interface requires some on-chip memory resource. Thus, using one external memory interface reduces the overall on-chip memory utilization.

- Modern FPGAs have a hierarchical structure, including multiple-chiplet FPGAs (see Section 4.1). Crossing between chiplets is expensive, resulting in low frequency [28]. Comparatively to a 2D grid, a 1D grid of PEs minimizes the number of crossings between chiplets. Thus, 1D PE scales to multiple chiplets better than 2D grids.

PE architecture. Figure 4.7(c) shows the internal architecture of a PE. Each PE has three local buffers, two for storing the incoming elements from input A and B and one for storing

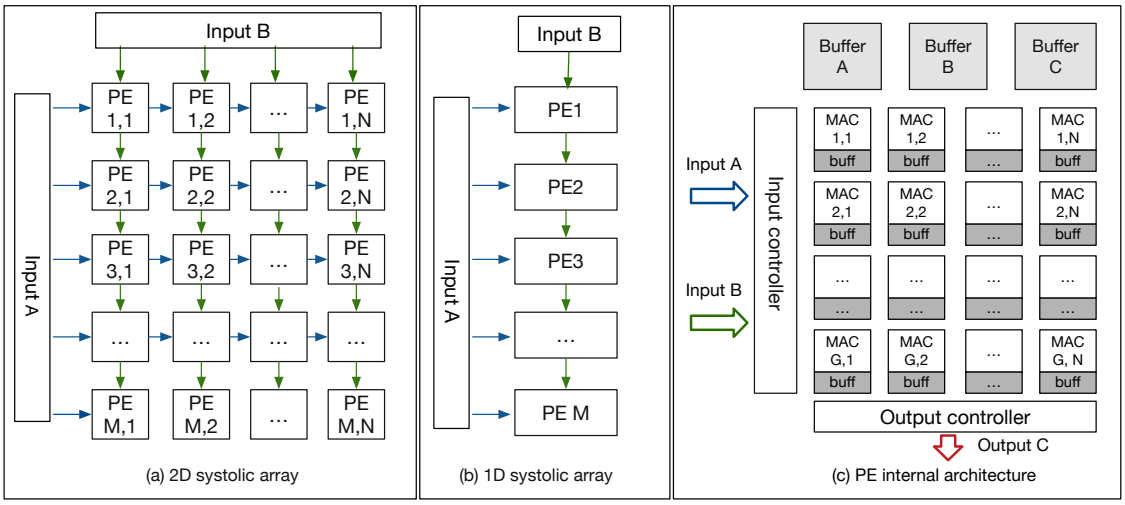


Figure 4.7: (a) A 2D structure of PEs for the GEMM unit. (b) A 1D structure of PEs for the GEMM unit. (c) The internal architecture of a PE unit.

partial products. Similar to PU units, we use double buffering techniques to allow data loading and computation to overlap. Each PE also has $G \times N$ MAC units. G is a design parameter and should match the group size in the pruning step. The N is the width of the systolic array. The input controller manages the incoming values and forwards them to the neighbors. The output controller is responsible for managing the output buffer and forwarding the result. Listing 4.3 shows the high-level overview of some of the operations in each PE. In each round, the PEs load a tile of the input data into its local buffers (Lines 18,19 in Listing 4.3). In addition to the local buffers to store the incoming tiles, the MAC units also have small buffers (listed with *valA, valB, and valC* in Listing 4.3). The tiles are loaded into the MAC's buffer before the computation (lines 21,22). All the inputs and output buffers are fully partitioned to allow parallel access to each element (lines 8,10,12, and 14). Similarly, the nested loop for performing multiply-accumulate (MAC) is fully unrolled (lines 24,26). This creates $N \times G$ MAC units. Since we skip the zero columns in A and the zero rows in B, the number of input tiles received by the PEs may vary. To notify the PE about the last tile in a round, we use an extra input (*EndStream*). If the bit (*end_i*) is set, it indicates the end of the input stream, while a zero indicates that more inputs are expected. Once all inputs have been processed, the output controller sends the result,

starting with sending the result received from other PEs (*OutputStreamI*). Next, we will describe how our design exploits sparsity.

4.2.3 Sparsity-aware Design

Our design utilizes sparsity in both inputs (feature map and weight). Sparsity is leveraged in three ways. First, we decrease the storage requirements by storing the weights in a compressed format. Second, we reduce the computation requirements by skipping computations involving zeros. Finally, we reduce the data transfers that involve zeros. Previous sparsity-aware designs used expensive hardware units such as CAM [52], prefix sums [42], and high fan-out memory units [30, 98]. None of these units are appropriate for FPGAs. In contrast, our sparsity-aware design does not require any expensive hardware components.

Our sparsity-aware design has two key features that make it suitable for FPGAs. First, the sparse format for storing weights is designed such that high-speed parallel access to the weight elements is possible with optimizations such as memory partitioning. Second, zeros are skipped outside the PEs and inside the GEMM input controller. As a result, the PE's design is simplified and unnecessary data traffic between PEs is avoided.

Sparse format for weights. Each PE in the GEMM unit receives G elements of the weight input (see Figure 4.7(c)). Thus, we divide each column of the weight matrix into blocks of G elements. The weights are accessed in column-major order by the GEMM unit. To keep the PEs in the GEMM active, we need high-bandwidth access to the weight blocks. By partitioning the weights into multiple memory units, elements can be accessed in parallel. Moreover, to reduce the storage requirement for the pruned weight, the weights need to be compressed. Unlike a dense representation, most standard formats are not suitable for data partitioning. Figure 4.8 demonstrates the inefficiency of using a CSC format for data partitioning. In this example, the elements are divided into blocks of size 2 (Figure 4.8(a)). Each block in a column is shown with a different color. Figure 4.8(b) shows how the non-

```
void PE (InputAStrm,InputBStrm,EndStrm,OutputStrmI,OutputStrmO,...)
{
  //PE input_controller buffers
  VECTOR_TYPE_A ABuffer [TILE_SIZE] ;
  VECTOR_TYPE_B BBuffer [TILE_SIZE] ;
  //MAC unit buffers
  int16_t valA [G] ;
  #pragma HLS ARRAY_PARTITION variable=valA complete dim=1
  int16_t valB [N] ;
  #pragma HLS ARRAY_PARTITION variable=valB complete dim=1
  int32_t valC [G][N] ;
  #pragma HLS ARRAY_PARTITION variable=valC complete dim=0
  int32_t prev [G][N] ;
  #pragma HLS ARRAY_PARTITION variable=prev complete dim=0
  for (unsigned cB = 0 ; cB < numcolB ; cB += N) {
    bool end_i = EndStrm.read() ;
    for (unsigned iteration = 0 ; end_i < 1 ; iteration++) {
      load_tile(InputAStrm,ABuffer);
      load_tile(InputBStrm,BBuffer);
      for (unsigned t = 0 ; t < TILE_SIZE ; t++) {
        load_mac_bufferA(t,ABuffer,valA);
        load_mac_bufferB(t,BBuffer,valB);
        for (unsigned r = 0 ; r < G ; r++) {
          #pragma HLS UNROLL
          for (unsigned c = 0 ; c < N ; c++){
            #pragma HLS UNROLL
            #pragma HLS PIPELINE II=1
            prev[r][c] = (iteration==0) ? 0 : valC[r][c] ;
            const int16_t mm = (valA[r] * valB[c]);
            valC[r][c] = mm + prev[r][c];
          }
        }
      }
      end_i = EndStrm.read() ;
    }
    //sending the output
    send_output(valC,OutputStrmI,OutputStrmO);
  }
}
```

Listing 4.3: The high-level HLS design for the PEs in the GEMM unit.

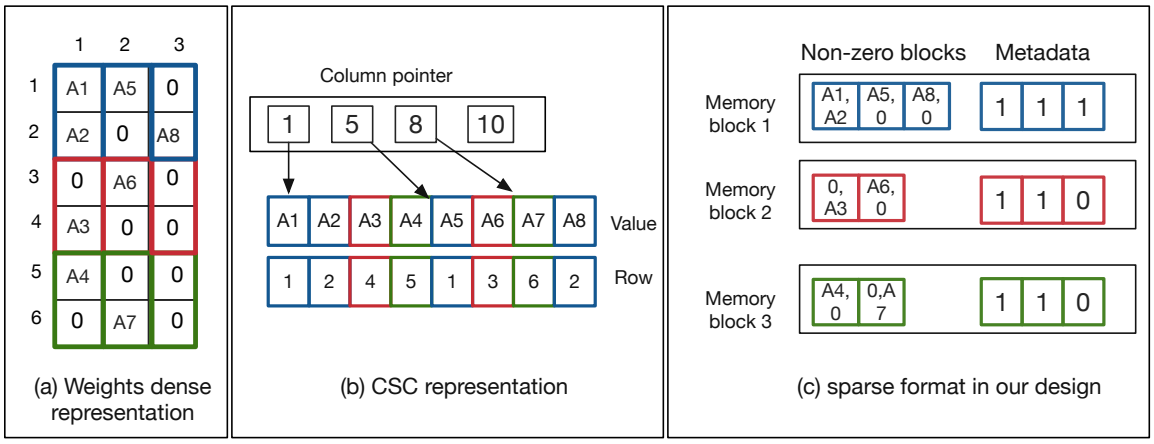


Figure 4.8: An illustration of our sparse format in comparison to the CSC format.

zero elements are stored in a CSC format. The CSC format uses a column pointer to identify the beginning of each column. The metadata used in a CSC format is insufficient to locate all the non-zero blocks in a column. To provide highly parallel access to non-zero elements and to allow compact storage of data, we developed our own custom format. Figure 4.8(c) shows our custom sparse format for storing the weights. In our sparse format, we store only the non-zero blocks. A block is considered zero if all the elements inside the block are zeros. The non-zero blocks are partitioned between multiple memory banks according to their position in a column. Each block contains one bit of metadata. A one indicates a non-zero block, and a zero indicates a zero block. Metadata for each block is also divided between memory banks similarly to values. We reduce metadata by encoding bitmaps in blocks rather than as individual elements. Figure 3.9 compares our sparse format with other sparse formats in terms of storage overhead. We apply group-wise pruning (Section 5.1) so that the group size matches the block size in our sparse format to increase the number of zero blocks and thus, reduce the overall storage. Finally, unlike applications in Chapter 2, using a custom sparse format and a preprocessing step is justified for CNN's inference since the weights remain unchanged throughout the inference task.

Skipping the computation and data transfers with zeros. The sparse weights do not change during the entire inference task. Prior work used various ways to organize the sparse weight and store them onto the FPGA on-chip memory prior to the computation. Unlike the zeros in the weights, the zeros in the feature map appear at run-time and are unpredictable. Hence, most FPGA designs only support sparsity for the weight input.

In Section 3.4.3, we discussed how a tall structure of PEs helps to avoid zeros by increasing the probability of finding zero blocks in the input feature map. We skip the zeros within the GEMM unit input controller rather than in the PEs. There are two advantages to skipping zeros in one central unit (i.e., the GEMM input controller). First, we avoid some zero-based data traffic between PEs. Second, it simplifies the PEs' design and eliminates unnecessary logic within each PE to detect and skip the zeros. In each step, a column of input A (weight) and a row of input B (feature map) arrive in the input controller. If either of the inputs (column of A or row of B) are zeros, we do not send that row and column to the GEMM unit. As a result, no expensive logic is required on the FPGA, and the operation can be pipelined. The major drawback of this approach for FPGAs is the high fanout of input controller. In our approach, the input controller is connected to all the PEs on the border of the GEMM unit.

4.2.4 Design Scalibility and Handling Various CNN Layers

In Section 3.4.4, we discussed how the dynamic reconfigurability of the GEMM unit is essential in efficiently executing CNN layers with different features. We defined two structures for the systolic array in the GEMM unit. One is a tall array where all the PEs are getting input from one main IM2COL unit. Alternatively, the PEs can be configured as multiple smaller GEMM units where each GEMM unit has its own IM2COL unit. The main IM2COL unit is used for the tall array configuration. For the other configuration, all IM2COL units are being used.

In addition to adapting to different CNN features, it is important that an FPGA design

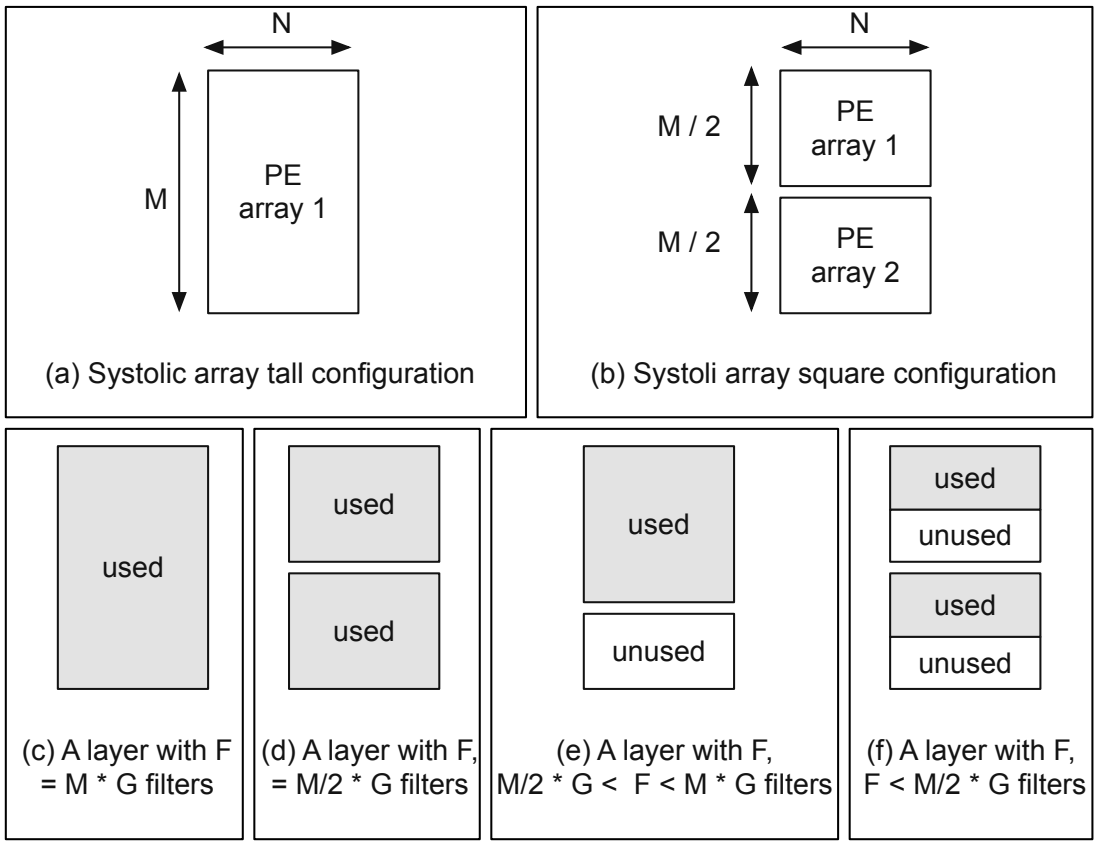


Figure 4.9: A demonstration of our design's flexibility in supporting CNN layers with a variety of filters. The variables F, M, N, and G indicate the number of filters, the systolic array height, the systolic array width, and the number of elements per PE, respectively.

be scalable (up or down) to suit FPGAs with different available resources. We can scale our design in two ways. First, the number of PEs in the GEMM unit can be chosen based on the available resources and the features of the layers in a CNN. Second, the available configuration for the GEMM unit can also be selected based on the FPGA resources. In our design in Chapter 3, the GEMM unit can be configured as four smaller arrays. This requires four IM2COL units for each smaller array. Due to the limited resources on the FPGA compared to ASIC, we use only two IM2COL units in our FPGA prototype. Figure 4.9(a-b) shows the two possible configurations for the GEMM unit in our FPGA design. If there are few filters in a CNN layer, using fewer IM2COL units may result in lower PE utilization. Figure 4.9(c-f) shows how different modes can be applied to layers with different numbers of filters. In this example, the systolic array can be used as a one tall array (Figure 4.9(a))

or two smaller arrays (Figure 4.9(b)). The value F indicates the number of filters in a CNN layer. The PEs in the arrays are fully or partially utilized depending on the value of F and the size of the systolic array.

4.3 Related Work on FPGA Accelerators for CNNs

In Section 3.5, we reviewed the ASIC design for accelerating sparse neural networks. Here, we primarily focus on FPGA accelerators for CNNs.

Sparsity-awareness design for FPGAs. There is a plethora of work on designing FPGA-based inference engines for dense CNNs [74, 75, 87, 96, 110, 132, 134]. Dense accelerators achieve near-perfect DPS efficiency (more than 95%) but in recent years, the focus has shifted towards developing sparse CNN accelerators. At a high level, sparse FPGA accelerators can be divided into two categories, one approach supports random pruning [85] and the other approach uses a structured pruning technique [64, 78, 151]. Random pruning has reduced the required MAC operations without compromising network accuracy [48], but they introduce irregularity that results in complex hardware, which is unsuitable for FPGAs with limited resources. Thus, the majority of FPGA accelerators use different variations of a structured pruning technique that suit their proposed dataflow and PE structure [75, 129]. Like our design, various studies present a hardware-software co-design framework that addresses irregular memory access issues for sparse computations on FPGAs [73, 75, 96]. They perform a regularization step in the software to enable high parallelism and data reusability for the FPGA. Orthogonal to these approaches, recent work generates a compressed model representation of the weights using orthogonal variable spreading factor (OVSF) binary codes during training which helps them to generate the weight *on-the-fly* during inference [73, 129].

Despite numerous ways of exploiting sparsity in weights, little research has been explored utilizing sparsity in the feature map. We learned in Section 3.5 that many ASIC designs have shown a significant potential for using sparsity in the feature map [30, 42, 98].

Most of these ASIC designs support the sparsity in both weights and feature map (dual-side sparsity) by introducing expensive hardware components such as a prefix sum for SparTen [42] and content addressable memory for ExTensor [52]). Thus, the proposed methods can not be directly used by an FPGA due to architecture-layout mismatch.

Supporting different CNN layers. CNNs have diverse layer shapes and sizes. The variety in CNN architecture highlights the need for flexibility when designing an accelerator. The flexibility can concern (1) functionally supporting different CNN layers with distinct features and (2) maintaining high efficiency across various layers in a CNN [124].

[99] and [37] suggest a *weight stationery* dataflow that requires all weights to be stored completely on the FPGA's on-chip memory. If the weight exceeds the on-chip resources of one FPGA, multiple FPGAs are deployed [37]. While some designs can support any CNN layer regardless of the on-chip memory resources of an FPGA, they may not perform well for certain layer shapes and sizes [78, 129]. To achieve high efficiency for various layers' features, [111] propose a flexible compilation framework to schedule matrix multiply and convolution operations of CNNs inference on the FPGA overlay. Similarly, [105] propose the cascaded interconnections that can achieve a very high-frequency design on FPGAs (650 MHz). The drawback of their method is that the FPGA must be reconfigured for each layer.

DNN accelerator generation for FPGAs. FPGA/ASIC-based DNN accelerators are in high demand, which led to the creation of automated DNN accelerator generation. For example, The DeepBurning [133] design automation tool uses pre-configured RTL modules to build DNN accelerators with customized design parameters. DNNBuilder [146] and FPDNN [44] provide end-to-end tools to automatically generate optimized FPGA-based accelerators from high-level DNN symbolic descriptions within Caffe/Tensorflow frameworks. Caffeine [143] is another automation tool to generate efficient FPGA design for CNNs. The tool takes into account the FPGA hardware parameters and generates the design based on the CNN layers. All these automation tools work only on a dense neu-

ral network. Incorporating sparse inputs increases the complexity of automatic hardware generation.

4.4 Summary

This chapter presents the FPGA version of our accelerator (SPOTS) for sparse CNN inference. Similar to the design in Chapter 2, we formulate the convolution as GEMM operation using an IM2COL operation. By offloading both IM2COL and GEMM units to the FPGA, we pipeline the two executions and avoid large data transfers between the CPU and the FPGA. Our design reduces the on-chip memory requirements for the IM2COL and GEMM units to better suit the FPGA with rather limited on-chip memory resources. On top of that, unlike most FPGA designs for sparse CNNs, we exploit the sparsity in both weights and feature maps. We do this without introducing any expensive hardware. The underlying principle of our sparsity-aware design is to use a group-wise pruning technique along with a processing step in software that allows us to store weights in a compressed format while allowing for highly parallel access to those weights. Finally, our design can scale up or down depending on the available resources on a target FPGA. This is in addition to our flexible design that can be configured based on the layers' features to achieve high PE utilization for various CNN layers. We evaluate our FPGA design along with the ASIC implementation in Chapter 5.

CHAPTER 5
EXPERIMENTAL EVALUATION OF OUR ASIC AND FPGA ACCELERATORS FOR SPARSE CONVOLUTIONAL NEURAL NETWORKS

Chapter 3 and Chapter 4 described a hardware/software solution for sparse convolutional neural networks designed for ASICs and FPGAs, respectively. This chapter evaluates both designs and compares them with general purpose architectures and other hardware accelerators in terms of performance and energy efficiency. Our analysis highlights some of the key aspects of both designs. In the first section of this chapter, we describe our experimental setup, including the systems we used for our evaluation, the benchmarks we used, and the features of the hardware we compared our design with (Section 5.1). Then, in Section 5.2, we evaluate different features of our FPGA and ASIC designs for sparse CNNs inference.

5.1 Experimental Methodology

SPOTS ASIC Prototype. A prototype of our design has been built in Verilog and synthesized using the FreePDK 45nm technology from Synopsys Design Compiler [121]. We achieve a frequency of 500 MHz with our design. There are no SRAM cells in FreePDK 45. Thus, we separately model the area and power of all the SRAM/DRAM using Cacti 7.0 [12]. Table 5.1 provides the parameters of the SPOTS prototype and the area breakdown for different components. We perform a cycle-accurate simulation of the RTL model of SPOTS in Verilog using Verilator. Using Synopsys's PowerPrime tool, we calculated the power usage of our design based on the traces from the RTL simulation. We ran our simulation for each layer individually. Weights are preprocessed and provided in our proposed sparse format. For the input feature map, we extracted each layer's data from the models in Caffe. Additionally, we developed an infrastructure to perform fast design space exploration and collect statistics.

Table 5.1: SPOTS ASIC design parameters and area.

Unit		Size	Area (mm2)
GEMM	#PE units	512	2.048
	Multiplier width	16 bits	
	Accumulator width	24 bits	
	Systolic array configurations	one (128×4) four (32×4)	
	PE's local buffers	2 KB	
IM2COL	#PU units	4	1.137
	Reserved buffers	32 KB	
	Other SRAM buffers	2 MB	
On-chip memory	Filter SRAM	1 MB	5.426
	Fmap SRAM	512 KB	
SPOTS total			8.611

Table 5.2: The CPU, GPU and FPGA configurations for SPOTS evaluation.

Platform	Compute units	Memory units	Technology
Intel Xeon E5-V3	4 cores, 3GHz	10 MB Smart Cache 32GB DDR4 (2666 Mhz)	22nm
Titan X Pascal	3584 cores, 1.53GHz	24GB of GDDR5	16nm

SPOTS FPGA Prototype. We built an end-to-end FPGA prototype of SPOTS (Chapter 4) using the Xilinx HLS tool and ran it on Alveo U200 card from Xilinx. Table 5.3 shows the FPGA specification. On Alveo U200, our FPGA design reaches a frequency of 154 MHz after place and route. Table 5.4 provides details on resource usage for two prototypes of FPGA design. One design only uses the GEMM unit as one large tall array with one IM2COL unit. The other design uses two IM2COL units that allow the GEMM unit to be configured as either a large tall array or two smaller GEMM units.

CPUs, GPUs used for our evaluation. We compare our design to CPUs and GPUs. In Table 5.2, we list the CPU and GPU we used for the evaluation. In our experiments, we used CPUs and GPUs manufactured with 22 nm and 16 nm cell technology, as opposed to SPOTS' 45 nm technology. We use Caffe to evaluate a variety of CNN architectures on a modern CPU and GPU. The Caffe framework uses IntelMKL for CPU computation and Nvidia's CUDA library, cuSparse, for GPU computation. Caffe also uses IM2COL +

Table 5.3: The FPGA configurations for SPOTS evaluation.

Platform	Compute units	Memory units
Xilinx Alveo-U200	1,182K logic elements 2280 DSP blocks	64 GB DDR4, 25-Mbits BRAM

Table 5.4: FPGA resource utilization and operating frequency for two versions of SPOTS on Alveo U200.

Application	Frequency	LUT	BRAM	URAM	FF	DSP
SPOTS-with-one-IM2COL	165Mhz	11%	24%	0%	10%	16%
SPOTS-with-two-IM2COL	154Mhz	35%	30%	3%	12%	24%

GEMM for its convolution layers, as we did. The energy consumed by the XEON CPU was measured using the Processor Counter Monitor (PCM). We measured GPU power consumption with NVIDIA System Management Interface (NVIDIA-SMI), which uses built-in sensors to query power consumption. According to NVIDIA, the reported data is accurate (*i.e.*, within $\pm$ 5 Watt).

ASICs used for our evaluation. We evaluated the performance and energy efficiency of SPOTS in comparison with other ASIC designs. Some designs support sparse inputs while others do not.

Eyeriss. For comparison, we use Eyeriss [21], an ASIC designed for accelerating sparse CNNs. The performance of Eyeriss was measured using the publicly available simulator [39]. An Eyeriss chip operates at a 200 MHz clock frequency and is fabricated at 65nm CMOS. Because we used a different cell technology (i.e., 45 nm) for SPOTS, we assume that the frequency of Eyeriss will be exactly equal to the frequency of SPOTS when reporting the execution time. Eyeriss was also configured to use the same number of MAC units and on-chip memory as SPOTS.

Gemmini. Gemmini [40] is a recent open-source full-stack DNN accelerator generator. Gemmini features a systolic array of processing elements (PEs), similar to SPOTS. Each PE performs dot products and accumulations. The PEs read the data from a local, explicitly

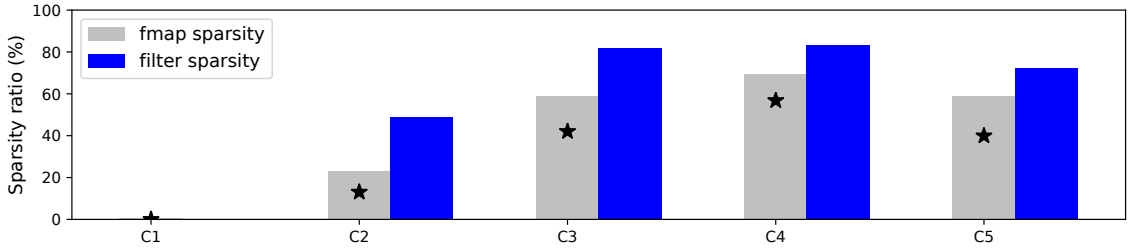
Table 5.5: Network characteristics, the top1, and top5 result accuracy, and the overall sparsity for the original (with no pruning), random pruning, and our structured pruning using the imagenet dataset. Weights and activations assume a data-type size of two bytes.

Model	#conv Layer	Baseline		Random pruning			Our structured pruning		
		Top1(%)	Top5(%)	Top1(%)	Top5(%)	Sparsity(%)	Top1(%)	Top5(%)	Sparsity(%)
AlexNet [68]	5	56.81	79.95	56.75	79.28	63.1	55.25	78.62	56.81
VGGNet [113]	13	68.27	88.36	68.21	88.25	62.8	67.18	88.16	27.48
GoogleNet [125]	57	68.92	89.14	68.42	88.85	68.25	66.22	87.53	25.12
ResNet [50]	53	72.71	90.66	72.4	90.58	60.51	69.71	89.30	31.45

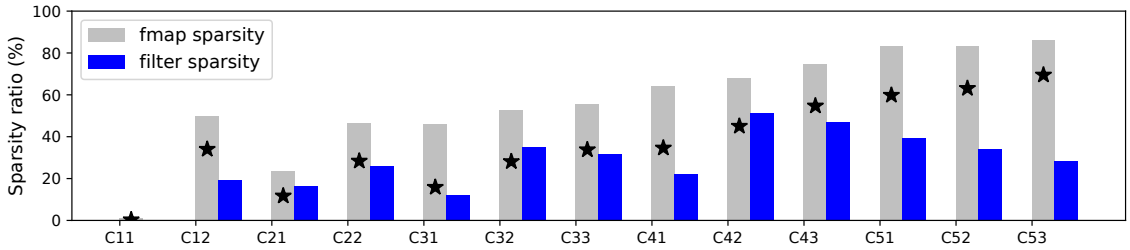
managed scratch-pad of banked SRAMs. We were unable to create a design with an exact number of PEs as SPOTS. We used tiles with 32×32 PEs for Gemmini, resulting in 1024 MAC units, which is $2\times$ more than our prototype. Gemmini's on-chip memory has been set to match SPOTS's on-chip memory.

Sparse-PE. Sparse-PE [102] is a recent hardware accelerator that supports sparse input for both feature maps and weights. This accelerator consists of multiple cores. Each core reads the inputs in a compressed form and performs three operations (*i.e.*, *selection*, *computation* and, *accumulation*) to generate the final result. We model their accelerator using the cycle counts and sparsity ratios reported in their paper. Their design natively supports CNN layers with a kernel size of 3. They perform kernel factorization on other kernel sizes. Besides, they only report the sparsity of the layers in AlexNet and VGGNet. Thus, our comparison is thus limited to those two networks. We used the same number of multiply units for SPOTS and Sparse-PE. Sparse-PE has an advantage since its design requires additional units for accumulation.

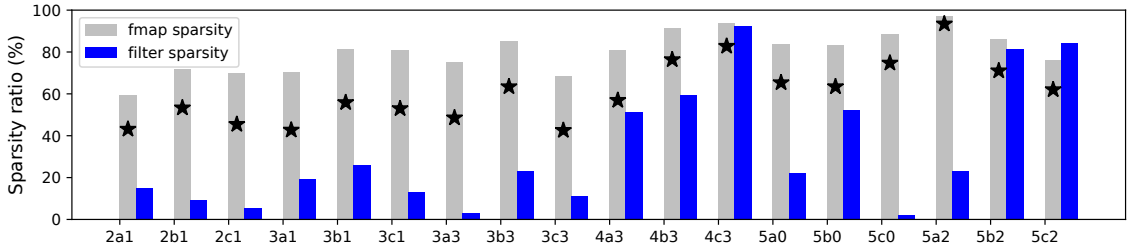
CNN architectures and pruning. We used four widely used CNN architectures: AlexNet [68], VGGNet-16 [113], GoogleNet [125], and Resnet-50 [50] to evaluate our prototype. We refer to VGGNet-16 and ResNet-50 as VGGNet and ResNet, respectively, throughout this section. Each of these CNN architectures is different in terms of layers, layer types, and sizes, as illustrated in Table 5.5. In all of our experiments, we used a batch size of one, which is the standard for an inference task. To train the networks, we used the input images from the Imagenet [31] dataset, a widely used dataset for image



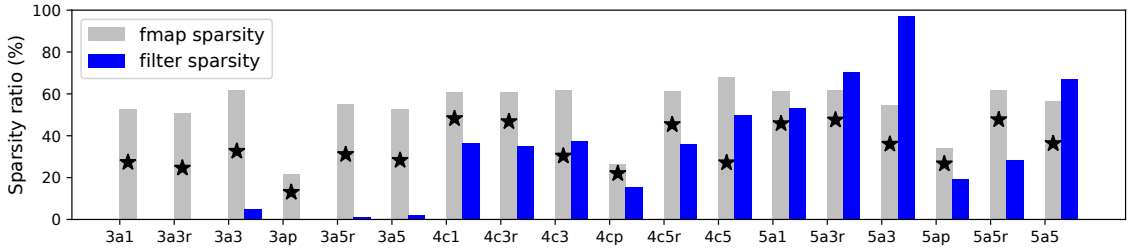
(a) AlexNet.



(b) VGGNet.



(c) ResNet.



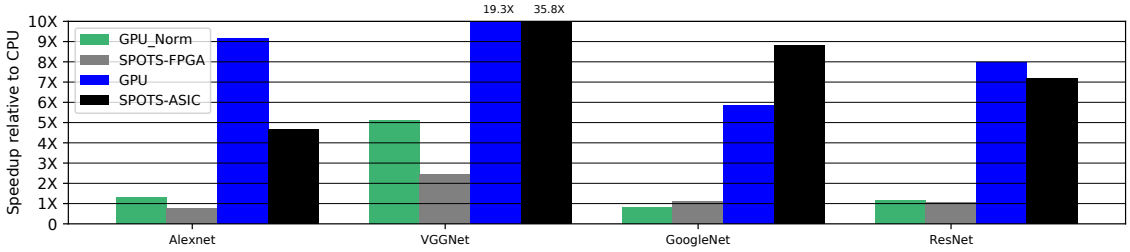
(d) GoogleNet.

Figure 5.1: Sparsity in the filters and input feature maps for AlexNet, VGGNet, ResNet, and GoogleNet. The $\star$ marker indicates the percentage of zeros in the output of the IM2COL transformation that is skipped on-the-fly by our design.

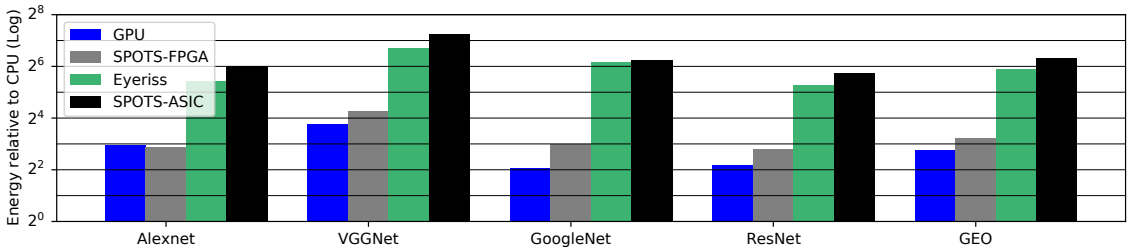
classification tasks. All four networks were pruned using the pruning algorithm based on Structure Sparsity Learning (SSL) [135]. SSL is generic and can be applied on different levels, including filters, channels, and shapes. We applied SSL at the shape level. Because our hardware exploits sparsity on a much finer granularity than shapes, we optimize SSL by pruning in a more fine-grained manner. Specifically, we zeroed the weights of elements of a shape that are below the threshold in some but not all cases. This generates zero blocks of a certain size (*i.e.*, the number of filters in the group). Figure 3.4(d) shows our group-wise pruning. Figure 5.1 displays the sparsity of the weights and input feature map for the layers of various CNN architectures. The sparsity varies between layers and networks. Lastly, we retrained the pruned network to regain its accuracy, which is the norm with pruning. Table 5.5 summarizes the accuracy and overall sparsity percentage for baseline (with no pruning), random pruning [49], and our structured pruning method. For accuracy results, we report the top-1 (i.e., the first prediction is correct) and the top-5 (i.e., the correct outcome is within the first 5 predicted values). Our pruned networks are within 1%-2% accuracy of the original model without pruning. Using structured pruning, one can achieve the same accuracy as a randomly pruned network with about 2× less sparsity. However, as was shown in Chapters 3 and 4, our structured pruning method simplifies the sparsity-awareness design significantly. Additionally, our sparse format outperforms the sparse format used by designs with random pruning (see Figure 3.9).

5.2 Experimental Evaluation of SPOTS

We begin by evaluating the ASIC and FPGA prototypes of SPOTS compared to general-purpose CPU and GPU implementations. We then compare our ASIC design with other ASIC designs for CNNs, including Eyeriss, Gemmini, Sparse-PE.



(a) Speedup with SPOTS



(b) Energy efficiency with SPOTS

Figure 5.2: (a) Speedup with SPOTS, GPU, and GPU implementations with the normalized number of MAC units over the CPU implementation as the baseline. (b) The energy efficiency of SPOTS and GPU implementations compared to a CPU baseline.

5.2.1 Comparing the Speedup of SPOTS ASIC and FPGA Prototypes with CPUs and GPUs

We compare the performance and energy efficiency of FPGA and ASIC prototypes of SPOTS to CPU and GPU executions. Figure 5.2a shows the speedup of SPOTS when compared with the CPU implementation. SPOTS ASIC implementation is 5×, 20×, 6×, and 8× faster than the CPU implementations using Intel MKL for AlexNet, VGGNet, GoogleNet, and ResNet, respectively. SPOTS achieves this speedup while operating at almost 6× less frequency than the CPU. For all networks except AlexNet, the FPGA prototype of SPOTS performs slightly better than the CPU implementation. Our FPGA prototype operates at 150 Mhz frequency, which is almost 20× less than the CPU's frequency. Figure 5.2a also shows the speedup of GPUs for the convolution layers over the CPU implementation. In comparison with GPUs, SPOTS' ASIC prototype is about 2× slower than GPU for AlexNet and VGGNet. However, it performs slightly better or is similar to GPU

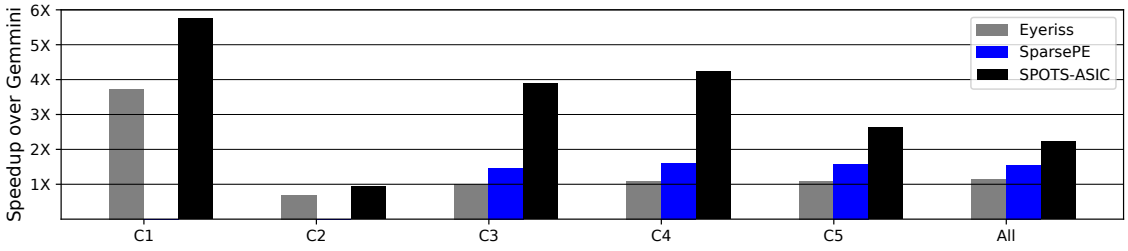
for GoogleNet and ResNet. VGGNet and AlexNet layers are relatively larger than the other two networks, creating larger matrices that favor GPUs with abundant MAC units, as opposed to SPOTS, which is designed for devices with limited power budgets. The first bar Figure 5.2a(a) shows the GPU performance when its number of MAC units is normalized to the number of MAC units in SPOTS. For the normalized number of MAC units, SPOTS outperforms the GPU on average by $6\times$. Finally, some prior work observed that the performance degrades for CPUs and GPUs when the sparse features are used when the networks are pruned randomly. We observed, however, that structured pruning helped CPU and GPU implementations achieve higher overall performance when using sparse linear algebra kernels.

5.2.2 Comparing the Energy Efficiency of SPOTS ASIC and FPGA Prototypes with CPUs and GPUs

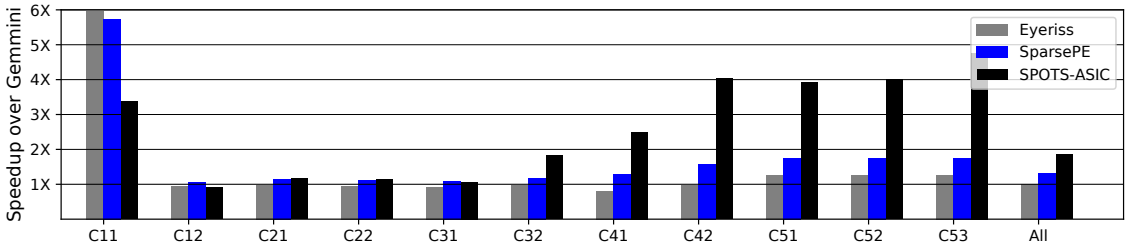
Figure 5.2b demonstrates the energy efficiency of the FPGA and ASIC prototypes of SPOTS and GPU implementations when compared to a CPU baseline for four CNNs. We did not include Gemmini energy results since their tool does not report the power consumption. The energy results include the off-chip memory accesses in this data. SPOTS ASIC implementation consumes $78\times$, $12\times$, and $1.4\times$ lesser energy than a CPU, a GPU, and Eyeriss, respectively. Compared to the CPU and GPU implementation, SPOTS' FPGA implementation is on average $9.3\times$ and $1.5\times$ more energy efficient, while it is less energy efficient than the two ASICs (SPOTS-ASIC and Eyeriss).

5.2.3 Comparing the Speedup of SPOTS ASIC Prototypes with Other ASIC Designs

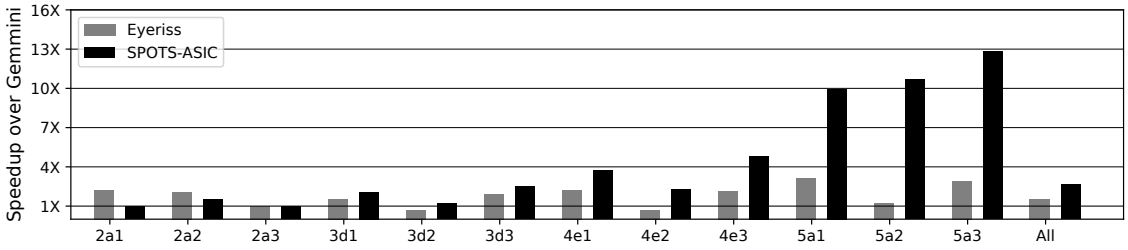
Figure 5.3 shows the speedup of SPOTS ASIC design, Eyeriss, and Sparse-PE relative to Gemmini for all four CNN architectures. Except for Gemmini, all other accelerators support sparse inputs. As some CNNs have many layers, only some of the layers are shown in the figures. Depending on where they appear in the network (top, middle, or bottom), the



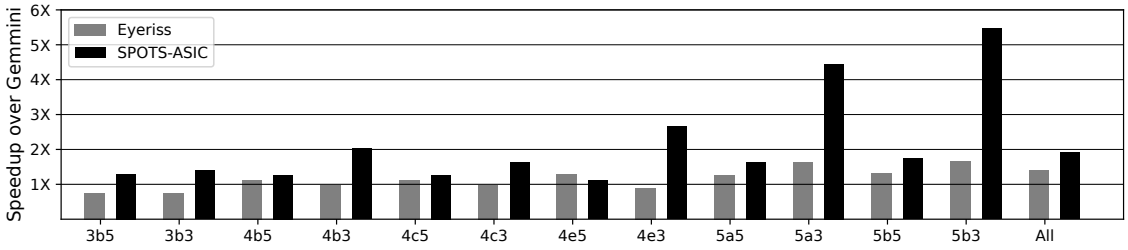
(a) AlexNet



(b) VGGNet



(c) ResNet



(d) GoogleNet

Figure 5.3: Speedup with SPOTS, Sparse-PE, and Eyeriss over Gemmini for four CNNs: AlexNet, VGGNet, ResNet, and GoogleNet. The figures show the speedup for selected layers from the top, middle, and bottom layers and the overall speedup (the last bar in each figure). For Sparse-PE, we only compare the speedup for AlexNet and VGGNet and layers with a kernel size of 3.

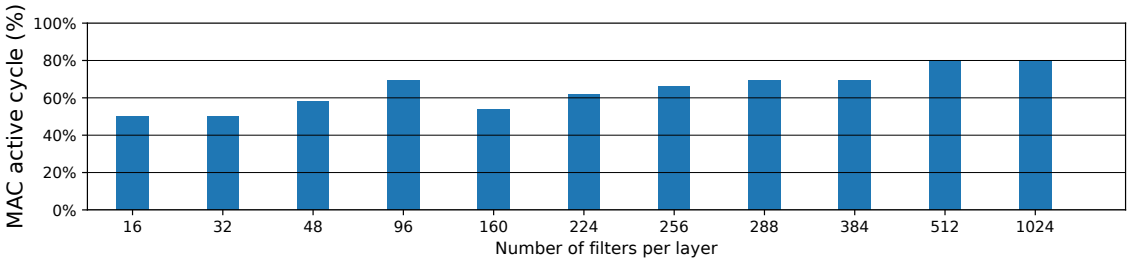
Table 5.6: Comparing the performance and efficiency of SPOTS ASIC design with different ASIC designs for sparse CNNs. All designs are scaled to 45nm technology.

Accelerator		SCNN [98]	NullHop [4]	SparTen [42]	**SPOTS**
Prunning Method		Random	N/A	Random	Structured
Bitwidth		16	16	16	16
Number of Multipliers		1024	1024	1024	512
Clock Frequency (MHz)		800	400	800	500
Core Area (mm2)		22.21	10.12	24.51	8.61
Throughput (Inference/s)	VGGNet	37.55	10.96	60.09	15.21
	AlexNet	479.92	N/A	767.88	249.79
Normalized Throughput (Inference/s)	VGGNet	2.93	6.85	3.75	15.21
	AlexNet	149.97	N/A	239.96	249.79
Power Efficiency (GOPS/Watt)	VGGNet	N/A	1357.51	440.84	469.33
	AlexNet	N/A	N/A	326.62	446.91
Area Efficiency (Inference/s/mm2)	VGGNet	1.69	1.08	2.45	1.76
	AlexNet	21.61	N/A	31.32	29.01

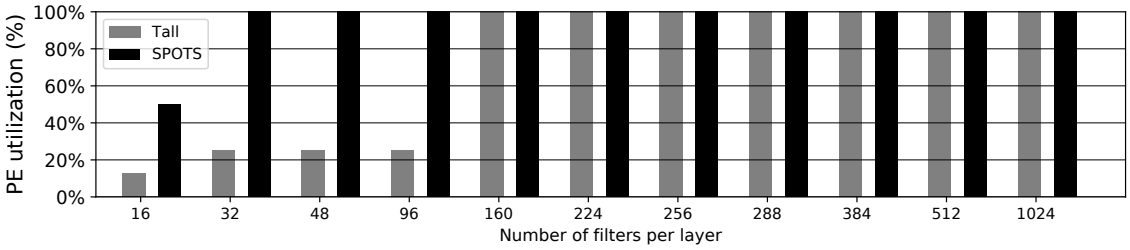
layers are sorted for each CNN architecture. Figure 5.3(a) shows the speedup for all layers in AlexNet. SPOTS ASIC design is nearly $2\times$ faster than Eyeriss and Gemmini on average. For the layers in the middle, where the sparsity ratio in the two inputs (/eg, weights and feature maps) is higher, SPOTS is nearly $4\times$ faster than Eyeriss and Gemmini. Besides the sparsity awareness that gives SPOTS an edge over Eyeriss and Gemmini, the bottom and middle layers have more filters that favor a tall systolic array. For Sparse-PE, we measure only the layers with a kernel size of 3 (see Section 5.1). For the measured layers, SPOTS is $1.8\times$ times faster than Sparse-PE.

Figure 5.3(b) reports the speedup for VGGNet. SPOTS ASIC design is, on average, $1.85\times$ and $1.86\times$ faster than Eyeriss and Gemmini. Similar to AlexNet, SPOTS achieves higher speedup with layers with more sparsity. SPOTS is slightly worse than Eyeriss in the first two layers since there are relatively few filters and the inputs are dense. We will demonstrate later in this section how the number of filters in a layer impacts PE utilization. SPOTS is, on average, $1.6\times$ faster than Sparse-PE. Because SPOTS uses data-reuse strategies and skips zero elements without requiring zero insertion and selection operations, it is more efficient than Sparse-PE.

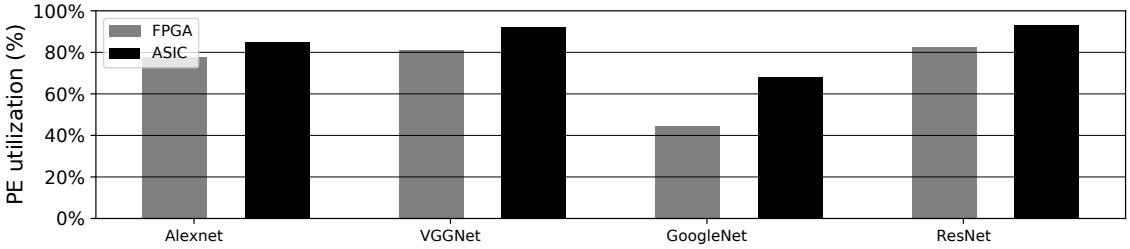
Figure 5.3(c) shows the speedup of SPOTS ASIC design over Eyeriss and Gemmini



(a) MAC utilization



(b) PE utilization

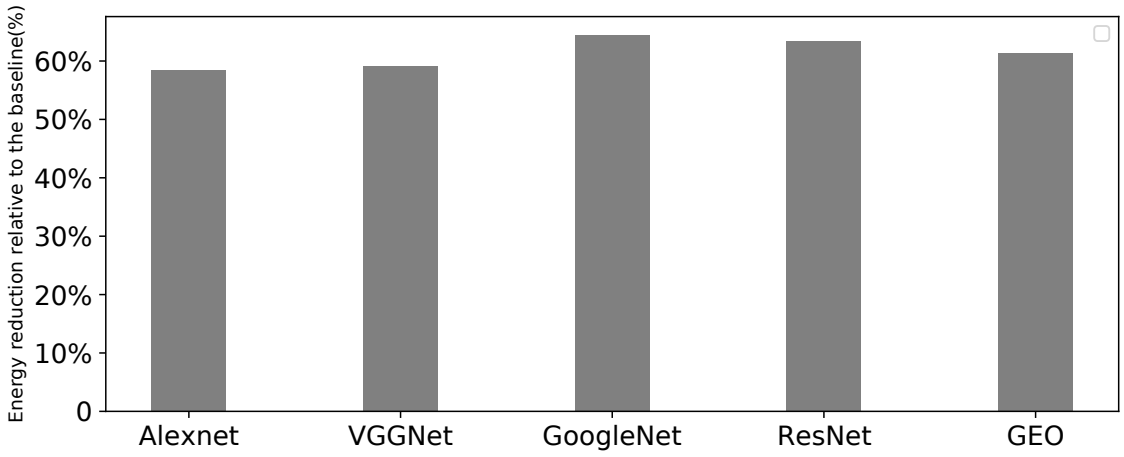


(c) FPGA Vs. ASIC PE utilization

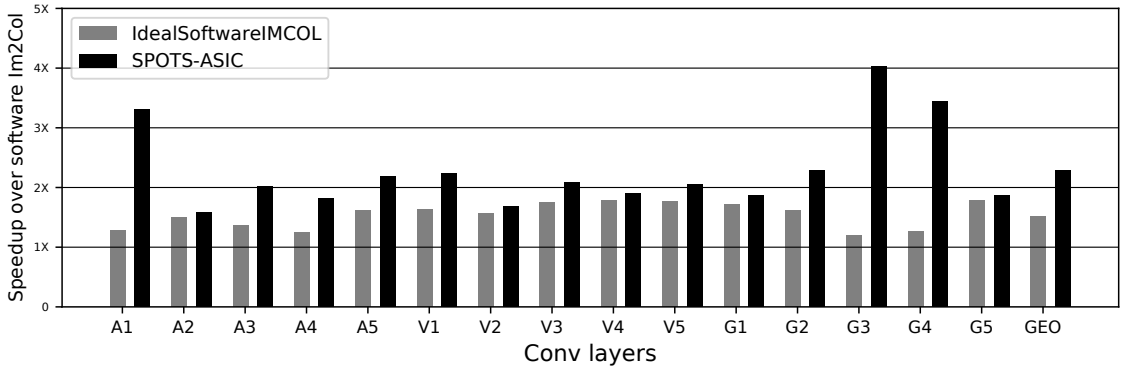
Figure 5.4: (a) MAC utilization (i.e., active cycles) for different filter sizes. (b) Comparing PE utilization of SPOTS and Tall systolic array (i.e., active PEs) for different filter sizes. (c) Comparing PE utilization of FPGA and ASIC prototypes for differnt CNNs.

for ResNet. On average, SPOTS is $1.77\times$ and $2.66\times$ faster than Eyeriss and Gemmini for ResNet. For layers where the weight and feature map sparsity are high, SPOTS is up to $8\times$ and $13\times$ faster than Eyeriss and Gemmini. As with VGGNet, SPOTS performs slightly worse than Eyeriss for the first eight layers in ResNet. This is because each layer in ResNet has a few filters. Hence, PEs are underutilized compared to layers in the middle or at the end of the network.

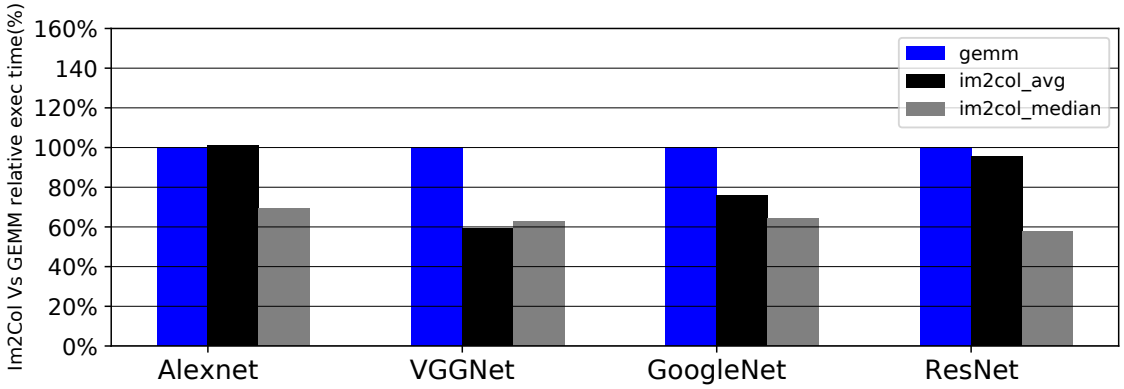
Figure 5.3(d) shows for GoogleNet, SPOTS ASIC design is $1.38\times$ and $1.91\times$ faster than Eyeriss and Gemmini, respectively. GoogleNet has more convolution layers with a



(a) IM2COL energy efficiency



(b) Hardware Vs. Software IM2COL



(c) IM2COL Vs. GEMM

Figure 5.5: (a) The reduction in energy consumption for the IM2COL unit of SPOTS over the baseline design. (b) Speedup with SPOTS over the software-based IM2COL as the baseline IM2COL design with no data reuse. (c) Fraction of the work performed by the IM2COL unit when compared to GEMM (i.e., GEMM bar is 100%). We report the average and the median for the IM2COL's work. When the mean exceeds the median, there will be instances where the IM2COL does more work compared to GEMM for some layers.

small number of filters that are not suitable for our tall array, in contrast to other CNN architectures. Overall, SPOTS provides less speedup for GoogleNet than the other three networks.

In Table 5.6, SPOTS ASIC design is compared with some of the prior sparse CNN accelerators in terms of its throughput, area, and power efficiency. All the accelerators are scaled to 45nm technology. We reported the exact number of multipliers that were reported in each paper. All three designs have twice as many multipliers as the SPOTS ASIC prototype. SCNN and NullHop clock frequencies are scaled to 45nm as in prior work [67]. For each design, the theoretical throughput varies depending on the clock frequency and the number of MAC units. The number of inference tasks completed per second is used to compare throughput. Table 5.6 shows the achieved throughput as well as the normalized throughput for each design. When the throughput is normalized (to have the same theoretical throughput as SPOTS), SPOTS outperforms all the other three accelerators. The Giga operations per second (GOPS) per Watt was used to compare power efficiency. SCNN does not report its power consumption. SPOTS ASIC design outperforms SparTen in power efficiency for both AlexNet and VGGNet. NullHop achieves the highest power efficiency while delivering lower throughput than SPOTS. Finally, SPOTS is comparable to SparTen and better than SCNN and NullHop in terms of area efficiency.

5.2.4 Performance Sensitivity to Different Layers' Shapes

The number and dimensions of filters differ for each layer of a CNN. SPOTS' dynamic reconfigurability allows the GEMM unit to be used either as a tall systolic array or as multiple small systolic arrays, adapting to a variety of shapes and filter sizes.

When there are a few filters (less than 128), the GEMM is configured with multiple small systolic arrays, each using a different IM2COL unit. All the PEs in the systolic array are active 100% of the time for all filter sizes other than 16 (see Figure 5.4b). By contrast, a tall systolic array without the enhancement we propose in Section 3.4.4 fails to achieve

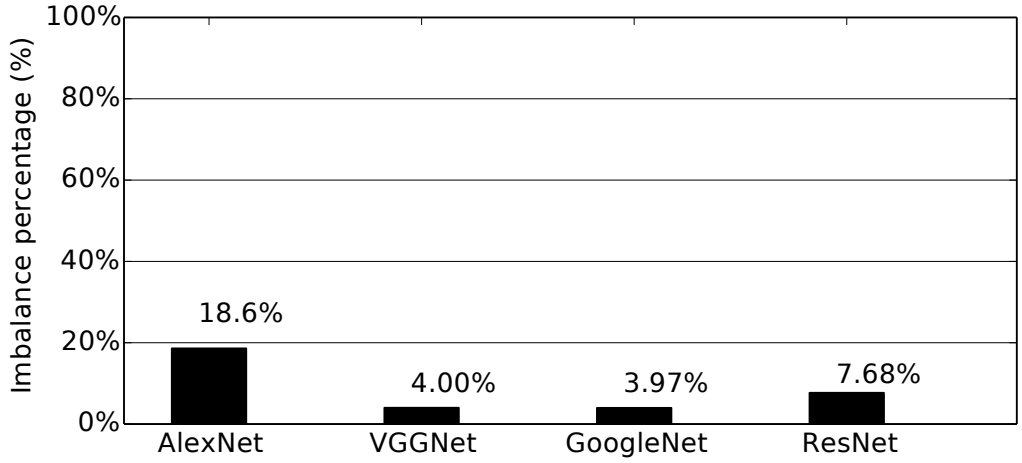


Figure 5.6: The load imbalance percentage in the pruned weights for AlexNet, VGGNet, ResNet, and GoogleNet are based on the metric defined in Equation 3.1.

full PE utilization for smaller filter sizes, as illustrated in Figure 5.4b. Figure 5.4a shows the utilization of the multiply-accumulate units in the PEs of the systolic array (*i.e.*, active cycles) when the layer has a specified number of filters (*i.e.*, x-axis reports the size of the filter). As the filter size increases, we assign more rows to a PE, which can fetch up to four elements per read. Therefore, there are more opportunities to keep multiply-accumulate units active in the PE (*i.e.*, almost 80% active cycles).

Comparing the PE Utilization for ASIC and FPGA Implementations FPGAs have limited resources, so a limited number of IM2COL units can be accommodated. As a result, the number of possible configurations is limited in the FPGA design compared to an ASIC implementation. Figure 5.4c illustrates the PE usage for ASIC and FPGA implementations for all four CNNs. The difference between PE utilization in the ASIC and FPGA implementations is less for AlexNet and VGGNet. In both networks, most layers have enough filters to allow the PEs to operate at full capacity with a tall configuration. Googlenet, on the other hand, has some layers with a limited number of filters in them that benefit from multiple smaller arrays with independent IM2COL units. As a result, the FPGA implementation with only two IM2COL units achieves a lower overall utilization than the ASIC design with four IM2COL units.

5.2.5 Performance and Energy Characterization of IM2COL Unit

Amount of work performed by IM2COL and GEMM units in SPOTS. As the IM2COL and the GEMM units are pipelined in SPOTS, ideally, the work done by the IM2COL unit and the GEMM unit should be balanced. Figure 5.5c shows the relative percentage of cycles where the IM2COL and GEMM units are active relative to the GEMM unit for each CNN architecture. As we report the active cycles relative to the GEMM unit, the bar for the GEMM unit is 100%. For AlexNet and ResNet, the average work performed by the IM2COL unit and the GEMM unit is almost identical (i.e., the work is balanced). In contrast, the total work in VGGNet is dominated by GEMM. The data suggest that adding more PEs to the GEMM unit may improve the overall VGGNet execution time. As the IM2COL unit is inactive because of the low bandwidth for AlexNet and ResNet, adding more PEs without increasing the bandwidth will not improve performance.

Energy efficiency from data reuse in the IM2COL unit. One of the key ideas in IM2COL's patch unit is to read the input feature map only once from the SRAM and reuse the data with local buffers. Figure 5.5a compares the reduction in energy consumption in our proposed IM2COL to a naive version of IM2COL that repeatedly accesses SRAMs without reusing data. Our mechanisms for reusing the input feature map in patch units result in an average of 60% less energy consumption in the IM2COL unit than the IM2COL unit without such a mechanism.

Comparing SPOTS IM2COL unit with a software IM2COL implementation. SPOTS has a hardware IM2COL unit that performs the IM2COL transformation on-the-fly. Figure 5.5b compares the speedup of using a hardware IM2COL unit compared to a software-based IM2COL as the baseline. In the baseline system, the hardware only performs GEMM while the CPU executes IM2COL. The figure also illustrates an ideal situation in which the software IM2COL and hardware GEMM computations overlap. Even when we provide an ideal scenario for the software IM2COL, SPOTS outperforms the ideal software-based IM2COL. SPOTS outperforms the baseline software IM2COL on average by 2.3×, show-

ing the benefits of our hardware IM2COL.

5.2.6 Load Imbalance in SPOTS

The non-uniform distribution of zero blocks in pruned weights causes a load imbalance in SPOTS. We quantified the load imbalance percentage among the PEs for the four studied sparse CNNs using the metric defined in Equation 3.1. For our analysis, we discarded all layers with sparsity ratios below 5%. A lower imbalance percentage indicates that PEs are idle less often due to the uneven distribution of non-zero blocks. All CNNs experience a very low load imbalance (less than 20%). Load imbalances are as low as 4% for VGGNet and GoogleNet, which shows that SPOTS load balancing strategies have been effective.

CHAPTER 6
CONCLUSION AND FUTURE DIRECTIONS

This dissertation presents a few hardware-software techniques to improve the performance and energy efficiency of two sparse linear algebra kernels (SpMV and SpGEMM) and a sparse convolutional neural network inference task. The general idea behind our approach is to utilize the software to reformat sparse data into a format that enables the hardware to perform the computation with a high degree of parallelism. First, we summarize the key contributions made by this dissertation, followed by directions for future research.

6.1 Dissertation Summary

As Moore's law approaches its end, specialized hardware is needed to continue improving computing performance and energy efficiency. Sparse computation is one of the areas where general purpose architectures fail to provide good performance. Due to the irregularity of memory accesses in sparse computation, a significant performance gap exists between dense and sparse kernels on CPUs and GPUs. In this dissertation, we present software-hardware solutions to some of the most important sparse problems, including two sparse linear algebra kernels (SpMV and SpGEMM) and sparse convolutional neural networks.

First, we make a case for a synergistic solution involving a CPU and an FPGA to accelerate sparse linear algebra kernels, including SpMV and SpGEMM. The CPU preprocesses the data and regularizes the memory accesses for the FPGA to perform the computation with a high degree of parallelism. We develop an intermediate representation that allows the software to communicate regularized data and scheduling decisions to the hardware. Decoupling the work done by the CPU and the FPGA in a coarse-grained fashion enables us to overlap their execution for higher performance. With an intermediate representation

and software preprocessing, we can support a wide range of sparse formats and data precisions. The speedups over mainstream libraries with our end-to-end system demonstrate that it is possible to achieve performance with FPGAs while being flexible to adapt to various formats.

Second, we propose an ASIC accelerator for sparse CNNs with a GEMM formulation of convolution using an IM2COL transformation. Both IM2COL and GEMM computations are performed in hardware and pipelined. Our proposed IM2COL unit avoids reading the feature map elements from external memory multiple times when the patches overlap. We add flexibility to the systolic array in the GEMM unit that allows us to achieve high PE utilization for CNN layers of varying shapes and sizes. Our design exploits sparsity in the input feature map and weights. We apply a group-wise pruning followed by a preprocessing step that transforms the pruned weights into a hardware-friendly compressed format. Using our structured pruning and sparse format, our sparsity-aware design does not require any expensive hardware unit to skip computations on zeros. Our design is faster and more energy efficient than state-of-the-art systolic array-based ASICs, CPU, and GPU implementations for sparse CNNs.

Finally, we present an FPGA design for accelerating sparse CNNs. Designing and manufacturing ASICs can take a long time and cost thousands of dollars. Instead, FPGAs are an alternative solution widely available in many data centers. We revised some components in our ASIC design, including the GEMM and IM2COL units, to adapt them to FPGAs. The key feature of our FPGA design for CNNs is that it can scale across different FPGAs with varying resource constraints. Furthermore, unlike many prior FPGA designs, our design exploits sparsity both in feature maps and weight inputs without requiring expensive hardware. Our FPGA design is more energy efficient than the CPU and GPU implementations for sparse CNN inference tasks.

6.2 Directions for Future Work

This dissertation makes a case for co-designing hardware and software to improve the performance and energy efficiency of some of the sparse computations, including SpMV, SpGEMM, and sparse CNNs. The ideas presented in this dissertation can be used to explore a variety of research directions. First, the ideas presented in REAP can be extended to other sparse linear algebra kernels. Another potential research direction is to explore methods to improve the programming productivity of our proposed design. There are two ways to do this. One approach is to develop an automation tool for generating software-hardware codes for various sparse formats and input precisions based on high-level specifications. Another approach for improving the programmability of FPGAs is to use other programming paradigms for HLS that better match the FPGA logic. Next, we discuss each of these potential directions.

6.2.1 Extending REAP to other Sparse Linear Algebra Kernels

The dissertation illustrates our software-hardware approach to accelerating sparse linear algebra kernels by demonstrating a design for SpMV and SpGEMM. REPA can be applied to other sparse linear algebra kernels, like sparse Cholesky factorization, which has many applications but has not been explored as widely as SpMV and SpGEMM. Sparse problems such as Cholesky factorization have different features than SpMV and SpGEMM kernels. For example, in Cholesky factorization, there are dependencies between different stages of computation. This means the computation of column K of the output cannot begin until all the dependencies from previous columns are resolved. This is in contrast to SpGEMM and SpMV kernels, where the result matrix can be computed independently and in any order. Another important difference between Cholesky factorization and SpGEMM is that the positions of non-zero elements are not fixed for all inputs as opposed to SpGEMM, where the non-zeros locations are known prior to the computation. Hence, Cholesky factoriza-

tion is more complicated than SpMV and SpGEMM kernels. Thus, accelerating Cholesky factorization requires new abstractions and new designs.

6.2.2 Synthesizing Hardware Accelerators for Sparse Problems

One of the key goals of our proposal was to support various sparse formats and data precisions by using software to process input with different sparse formats and then transform it into an intermediate format for the FPGA to perform the computation. We demonstrate the versatility of our design by supporting three sparse formats (CSR, ELL, and DIA) and three different data precisions (Float, Int16, Int8). A potential future work is to build a framework that can automatically synthesize software and hardware codes for different sparse formats and data precisions using high-level specifications. Some of the abstractions proposed by REAP can be used to simplify the task. Using an intermediate representation between the CPU and the FPGA can minimize the FPGA design. Thus, the main challenge is synthesizing the CPU's software side to generate the preprocessing task automatically.

6.2.3 Exploring New Programming Languages for Sparse Kernels on FPGAs

We used Xilinx HLS tool to build our FPGA designs. In Xilinx HLS, the programmer expresses the design using the C++ programming language with additional directives (pragmas). Mapping imperative code onto hardware structures requires complex heuristics. The irregularity introduced by the sparse problem exacerbates the problem of designing FPGA accelerators. One research direction is to explore other programming models that fit FPGAs better. One potential example of such a programming model is the Communicating Sequential Processes (CSP). The regularization step in REAP, which is done in software, allows computations to be regularized on the FPGA. FPGA design in REAP consists of a number of processing elements that are interconnected and communicate a stream of data through the channels. This model of design can be expressed very well using a CSP model.

BIBLIOGRAPHY

[1] 2014. cuSPARSE Library. https://developer.nvidia.com/cuSPARSE.

[2] 2015. S. Dalton, N. Bell, L. Olson, and M. Garland. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. Version 0.5.1.

[3] 2018. Intel math kernel library reference manual. Technical Report. http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf

[4] Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, Antonio Rios-Navarro, Ricardo Tapiador-Morales, Iulia-Alexandra Lungu, Moritz B. Milde, Federico Corradi, Alejandro Linares-Barranco, Shih-Chii Liu, and Tobi Delbruck. 2019. NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps. *IEEE Transactions on Neural Networks and Learning Systems* 30, 3 (2019), 644–656. https://doi.org/10.1109/TNNLS.2018.2852335

[5] K. Akbudak and C. Aykanat. 2017. Exploiting Locality in Sparse Matrix-Matrix Multiplication on Many-Core Architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 8 (Aug 2017).

[6] Hasan Metin Aktulga, Md. Afibuzzaman, Samuel Williams, Aydın Buluç, Meiyue Shao, Chao Yang, Esmond G. Ng, Pieter Maris, and James P. Vary. 2017. A High Performance Block Eigensolver for Nuclear Configuration Interaction Calculations. *IEEE Transactions on Parallel and Distributed Systems* 28, 6 (2017), 1550–1563. https://doi.org/10.1109/TPDS.2016.2630699

[7] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-Pragmatic Deep Neural Network Computing. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 382–394.

[8] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 13 pages. https://doi.org/10.1145/3007787.3001138

[9] Bahar Asgari, Ramyad Hadidi, Joshua Dierberger, Charlotte Steinichen, Amaan Marfatia, and Hyesoon Kim. 2021. Copernicus: Characterizing the Performance Implications of Compression Formats Used in Sparse Workloads. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 1–12. https://doi.org/10.1109/IISWC53511.2021.00012

[10] Brett W. Bader and Tamara G. Kolda. 2007. Efficient MATLAB Computations with Sparse and Factored Tensors. 30, 1 (dec 2007), 205–231. https://doi.org/10.1137/060676489

[11] Brett W. Bader and Tamara G. Kolda. 2007. Efficient MATLAB Computations with Sparse and Factored Tensors. *SIAM J. Sci. Comput.* 30, 1 (Dec. 2007), 205–231. https://doi.org/10.1137/060676489

[12] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article 14 (June 2017), 25 pages. https://doi.org/10.1145/3085572

[13] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Limsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R Clint Whaley. 2002. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2 (jun 2002), 135–151. https://doi.org/10.1145/567806.567807

[14] A. Buluc and J. R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *Proceedings of IPDPS*. 1–11.

[15] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel Sparse Matrix-vector and Matrix-transpose-vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures* (Calgary, AB, Canada) *(SPAA '09)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/1583991.1584053

[16] A. Buluç, S. Williams, L. Oliker, and J. Demmel. 2011. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. In *2011 IEEE International Parallel Distributed Processing Symposium*.

[17] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[18] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-Based Graph Processing Framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 69–80. https://doi.org/10.1145/3431920.3439290

[19] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 609–622. https://doi.org/10.1109/MICRO.2014.58

[20] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 367–379. https://doi.org/10.1109/ISCA.2016.40

[21] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[22] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276493

[23] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural Language Processing (Almost) from Scratch. *J. Mach. Learn. Res.* 12 (nov 2011), 2493–2537.

[24] Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. 2006. Platform-Based Behavior-Level and System-Level Synthesis. In *2006 IEEE International SOC Conference*. 199–202. https://doi.org/10.1109/SOCC.2006.283880

[25] J. Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *2006 43rd ACM/IEEE Design Automation Conference*. 433–438. https://doi.org/10.1145/1146909.1147025

[26] Kincaid David R., C. Oppe Thomas, and Young David M. 1989. *ITPACKV 2D User's Guide*.

[27] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages.

[28] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. 2020. Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 244–254. https://doi.org/10.1145/3373087.3375296

[29] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. 2019. Bit-Tactical: A Software/Hardware Approach to Exploiting Value and Bit Sparsity in Neural Networks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 749–763. https://doi.org/10.1145/3297858.3304041

[30] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo Yuan. 2021. GoSPA: An Energy-efficient High-performance Globally Optimized SParse Convolutional Neural Network Accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1110–1123. https://doi.org/10.1109/ISCA52012.2021.00090

[31] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[32] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (1974), 256–268. https://doi.org/10.1109/JSSC.1974.1050511

[33] Luiz DeRose, Bill Homer, and Dean Johnson. 2007. Detecting Application Load Imbalance on High End Massively Parallel Systems. In *Proceedings of the 13th International Euro-Par Conference on Parallel Processing* (Rennes, France) *(Euro-Par'07)*. Springer-Verlag, Berlin, Heidelberg, 150–159.

[34] Richard Dorrance, Fengbo Ren, and Dejan Marković. 2014. A Scalable Sparse Matrix-Vector Multiplication Kernel for Energy-Efficient Sparse-Blas on FPGAs. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '14)*. Association for Computing Machinery, New York, NY, USA, 161–170. https://doi.org/10.1145/2554688.2554785

[35] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. 2022. High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 54–64. https://doi.org/10.1145/3490422.3502368

[36] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 92–104. https://doi.org/10.1145/2749469.2750389

[37] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA.2018.00012

[38] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt. 2014. A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 36–43. https://doi.org/10.1109/FCCM.2014.23

[39] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 807–820. https://doi.org/10.1145/3297858.3304014

[40] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. arXiv:1911.09925 [cs.DC]

[41] Joachim Georgii and Rudiger Westermann. 2010. A streaming approach for sparse matrix products and its application in Galerkin multigrid methods. *Electronic Transactions on Numerical Analysis* 37, 263-275 (2010), 3–5.

[42] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 151–165. https://doi.org/10.1145/3352460.3358291

[43] P. Grigoras, P. Burovskiy, E. Hung, and W. Luk. 2015. Accelerating SpMV on FPGAs by Compressing Nonzero Values. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 64–67. https://doi.org/10.1109/FCCM.2015.30

[44] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 152–159. https://doi.org/10.1109/FCCM.2017.25

[45] Udit Gupta, Brandon Reagen, Lillian Pentecost, Marco Donato, Thierry Tambe, Alexander M. Rush, Gu-Yeon Wei, and David Brooks. 2019. MASR: A Modular Accelerator for Sparse RNNs. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 1–14. https://doi.org/10.1109/PACT.2019.00009

[46] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (Sept. 1978), 250–269. https://doi.org/10.1145/355791.355796

[47] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 75–84. https://doi.org/10.1145/3020078.3021745

[48] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 243–254. https://doi.org/10.1109/ISCA.2016.30

[49] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations (ICLR)* (2016).

[50] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[51] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices. In *Proceedings of the 34th ACM International Conference on Supercomputing* (Barcelona, Spain) *(ICS '20)*. Association for Computing Machinery, New York, NY, USA, Article 19, 12 pages. https://doi.org/10.1145/3392717.3392751

[52] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275

[53] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceeding of the Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, New York, NY, USA, 319–333.

[54] R. Hojabr, A. Sedaghati, A. Sharifian, A. Khonsari, and A. Shriraman. 2021. SPAGHETTI: Streaming Accelerators for Highly Sparse GEMM on FPGAs. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[55] Jong Hoon Shin, Ali Shafiee, Ardavan Pedram, Hamzah Abdel-Aziz, Ling Li, and Joseph Hassoun. 2022. Griffin: Rethinking Sparse Optimization for Deep Learning Architectures. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 861–875. https://doi.org/10.1109/HPCA53966.2022.00068

[56] Chao-Tsung Huang, Yu-Chun Ding, Huan-Ching Wang, Chi-Wen Weng, Kai-Ping Lin, Li-Wei Wang, and Li-De Chen. 2019. ECNN: A Block-Based and Highly-Parallel CNN Accelerator for Edge Inference. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 182–195. https://doi.org/10.1145/3352460.3358263

[57] S. Huang, C. Pearson, R. Nagi, J. Xiong, D. Chen, and W. Hwu. 2019. Accelerating Sparse Deep Neural Networks on FPGAs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[58] Bassam A Hussein and Ahmed A Shabana. 2011. Sparse matrix implicit numerical integration of the Stiff differential/algebraic equations: Implementation. *Nonlinear Dynamics* 65, 4 (2011), 369–382.

[59] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization framework for sparse matrix kernels. *The International Journal of High Performance Computing Applications* 18, 1 (2004), 135–158.

[60] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[61] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. SMASH: Co-designing Software Compression and

Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In *Proceedings of MICRO '52* (Columbus, OH, USA). 15 pages.

[62] Hyeong-Ju Kang. 2020. Accelerator-Aware Pruning for Convolutional Neural Networks. *IEEE Transactions on Circuits and Systems for Video Technology* 30, 7 (2020), 2093–2103. https://doi.org/10.1109/TCSVT.2019.2911674

[63] Hyeong-Ju Kang. 2020. Accelerator-Aware Pruning for Convolutional Neural Networks. *IEEE Trans. Cir. and Sys. for Video Technol.* 30, 7 (jul 2020), 2093–2103. https://doi.org/10.1109/TCSVT.2019.2911674

[64] Hyeong-Ju Kang. 2020. Accelerator-Aware Pruning for Convolutional Neural Networks. *IEEE Transactions on Circuits and Systems for Video Technology* 30, 7 (2020), 2093–2103. https://doi.org/10.1109/TCSVT.2019.2911674

[65] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John Owens, Marcin Zalewski, Tim Mattson, and José Moreira. 2016. Mathematical foundations of the GraphBLAS. 1–9. https://doi.org/10.1109/HPEC.2016.7761646

[66] S. Kestur, J. D. Davis, and E. S. Chung. 2012. Towards a Universal FPGA Matrix-Vector Multiplication Architecture. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 9–16. https://doi.org/10.1109/FCCM.2012.12

[67] Dae Hyun Kim and Sung Kyu Lim. 2015. *Impact of TSV and Device Scaling on the Quality of 3D ICs*. Springer New York, New York, NY, 1–22. https://doi.org/10.1007/978-1-4939-2163-8_1

[68] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (May 2017), 84–90. https://doi.org/10.1145/3065386

[69] H.T. Kung, Bradley McDanel, and Sai Qian Zhang. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 821–834. https://doi.org/10.1145/3297858.3304028

[70] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. 2019. Heterogeneous Dataflow Accelerators for Multi-DNN Workloads. arXiv:arXiv:1909.07437

[71] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. *SIGPLAN Not.* 53, 2 (March 2018), 461–475. https://doi.org/10.1145/3296957.3173176

[72] Edward L. Hauck. 1986. Data compression using run length encoding and statistical encoding. (dec 1986).

[73] Bing Li, Wei Wen, Jiachen Mao, Sicheng Li, Yiran Chen, and Hai Li. 2018. Running sparse and low-precision neural network: When algorithm meets hardware. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 534–539. https://doi.org/10.1109/ASPDAC.2018.8297378

[74] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–9. https://doi.org/10.1109/FPL.2016.7577308

[75] Sicheng Li, Wei Wen, Yu Wang, Song Han, Yiran Chen, and Hai Li. 2017. An FPGA Design Framework for CNN Sparsification and Acceleration. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 28–28. https://doi.org/10.1109/FCCM.2017.21

[76] Xiaoye S. Li. 2005. An Overview of SuperLU: Algorithms, Implementation, and User Interface. *ACM Trans. Math. Softw.* 31, 3 (sep 2005), 302–325. https://doi.org/10.1145/1089014.1089017

[77] C. Y. Lin, Z. Zhang, N. Wong, and H. K. So. 2010. Design space exploration for sparse matrix-matrix multiplication on FPGAs. In *2009 International Conference on Field-Programmable Technology*. 369–372. https://doi.org/10.1109/FPT.2010.5681425

[78] Linqiao Liu and Stephen Brown. 2021. Leveraging Fine-grained Structured Sparsity for CNN Inference on Systolic Array Architectures. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 301–305. https://doi.org/10.1109/FPL53798.2021.00060

[79] W. Liu and B. Vinter. 2014. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*.

[80] Weifeng Liu and Brian Vinter. 2015. A Framework for General Sparse Matrix-matrix Multiplication on GPUs and Heterogeneous Processors. *J. Parallel Distrib. Comput.* 85, C (Nov. 2015), 47–61. https://doi.org/10.1016/j.jpdc.2015.06.010

[81] Zhi-Gang Liu, Paul N. Whatmough, and Matthew Mattina. 2020. Sparse Systolic Tensor Array for Efficient CNN Hardware Acceleration. arXiv:2009.02381 [cs.AR]

[82] Zhi-Gang Liu, Paul N. Whatmough, and Matthew Mattina. 2020. Systolic Tensor Array: An Efficient Structured-Sparse GEMM Accelerator for Mobile CNN Inference. *IEEE Computer Architecture Letters* 19, 1 (2020), 34–37. https://doi.org/10.1109/LCA.2020.2979965

[83] Zhi-Gang Liu, Paul N. Whatmough, Yuhao Zhu, and Matthew Mattina. 2021. S2TA: Exploiting Structured Sparsity for Energy-Efficient Mobile CNN Acceleration. https://doi.org/10.48550/ARXIV.2107.07983

[84] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. 2021. Demystifying the Memory System of Modern Datacenter FPGAs for Software Programmers through Microbenchmarking. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 105–115. https://doi.org/10.1145/3431920.3439284

[85] Liqiang Lu, Jiaming Xie, Ruirui Huang, Jiansong Zhang, Wei Lin, and Yun Liang. 2019. An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 17–25. https://doi.org/10.1109/FCCM.2019.00013

[86] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 553–564. https://doi.org/10.1109/HPCA.2017.29

[87] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 45–54. https://doi.org/10.1145/3020078.3021736

[88] K. Matam, S. R. Krishna Bharadwaj Indarapu, and K. Kothapalli. 2012. Sparse matrix-matrix multiplication on modern architectures. In *2012 19th International Conference on High Performance Computing*. 1–10. https://doi.org/10.1109/HiPC.2012.6507483

[89] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr. 2017. Fine-grained accelerators for sparse machine learning workloads. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[90] Huiyu Mo, Leibo Liu, Wenjing Hu, Wenping Zhu, Qiang Li, Ang Li, Shouyi Yin, Jian Chen, Xiaowei Jiang, and Shaojun Wei. 2020. TFE: Energy-efficient Transferred Filter-based Engine to Compress and Accelerate Convolutional Neural Networks. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 751–765. https://doi.org/10.1109/MICRO50266.2020.00067

[91] Alistair Moffat and Justin Zobel. 1992. Parameterised Compression for Sparse Bitmaps. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (Copenhagen, Denmark)

(SIGIR '92). Association for Computing Machinery, New York, NY, USA, 274–285. https://doi.org/10.1145/133160.133210

[92] Gordon E. Moore. 2006. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter* 11, 3 (2006), 33–35. https://doi.org/10.1109/N-SSC.2006.4785860

[93] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable Accelerator Design with Time-Sensitive Affine Types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 393–407. https://doi.org/10.1145/3385412.3385974

[94] NVIDIA, Peter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. (2020). https://developer.nvidia.com/cuda-toolkit

[95] Declan O'Loughlin, Aedan Coffey, Frank Callaly, Darren Lyons, and Fearghal Morgan. 2014. Xilinx Vivado High Level Synthesis: Case studies. In *25th IET Irish Signals Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014)*. 352–356. https://doi.org/10.1049/cp.2014.0713

[96] Adam Page, Ali Jafari, Colin Shea, and Tinoosh Mohsenin. 2017. SPARCNet: A Hardware Accelerator for Efficient Deployment of Sparse Convolutional Networks. *J. Emerg. Technol. Comput. Syst.* 13, 3, Article 31 (may 2017), 32 pages. https://doi.org/10.1145/3005448

[97] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 724–736.

[98] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 27–40. https://doi.org/10.1145/3079856.3080254

[99] Jinhwan Park and Wonyong Sung. 2016. FPGA based implementation of deep neural networks using on-chip memory only. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 1011–1015. https://doi.org/10.1109/ICASSP.2016.7471828

[100] Eric Qin, Geonhwa Jeong, William Won, Sheng-Chun Kao, Hyoukjun Kwon, Sudarshan Srinivasan, Dipankar Das, Gordon E. Moon, Sivasankaran Rajamanickam,

and Tushar Krishna. 2021. Extending Sparse Tensor Accelerators to Support Multiple Compression Formats. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1014–1024. https://doi.org/10.1109/IPDPS49936.2021.00110

[101] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[102] Mahmood Azhar Qureshi and Arslan Munir. 2021. Sparse-PE: A Performance-Efficient Processing Engine Core for Sparse Convolutional Neural Networks. *IEEE Access* 9 (2021), 151458–151475. https://doi.org/10.1109/ACCESS.2021.3126708

[103] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 267–278. https://doi.org/10.1109/ISCA.2016.32

[104] Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[105] Ananda Samajdar, Tushar Garg, Tushar Krishna, and Nachiket Kapre. 2019. Scaling the Cascades: Interconnect-Aware FPGA Implementation of Machine Learning Problems. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 342–349. https://doi.org/10.1109/FPL.2019.00061

[106] Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. 2013. Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi. *CoRR* abs/1302.1078 (2013). arXiv:1302.1078 http://arxiv.org/abs/1302.1078

[107] Robert Schone, Daniel Hackenberg, and Daniel Molka. 2012. Memory Performance at Reduced CPU Clock Speeds: An Analysis of Current x86-64 Processors. In *2012 Workshop on Power-Aware Computing and Systems (HotPower 12)*. USENIX Association, Hollywood, CA.

[108] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. 2019. Laconic Deep Learning Inference Acceleration. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 304–317.

[109] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep

neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783720

[110] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 535–547. https://doi.org/10.1145/3079856.3080221

[111] Runbin Shi, Yuhao Ding, Xuechao Wei, He Li, Hang Liu, Hayden K.-H. So, and Caiwen Ding. 2020. FTDL: A Tailored FPGA-Overlay for Deep Learning with High Scalability. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218581

[112] Gil Shomron, Tal Horowitz, and Uri Weiser. 2019. SMT-SA: Simultaneous Multithreading in Systolic Arrays. *IEEE Computer Architecture Letters* 18, 2 (2019), 99–102. https://doi.org/10.1109/LCA.2019.2924007

[113] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV]

[114] Mohammadreza Soltaniyeh, Veronica Lagrange Moutinho Dos Reis, Matt Bryson, Xuebin Yao, Richard P. Martin, and Santosh Nagarakatte. 2022. Near-Storage Processing for Solid State Drive Based Recommendation Inference with SmartSSDs®. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering* (Beijing, China) *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 177–186. https://doi.org/10.1145/3489525.3511672

[115] Mohammadreza Soltaniyeh, Richard P. Martin, and Santosh Nagarakatte. 2020. Synergistic CPU-FPGA Acceleration of Sparse Linear Algebra. arXiv:2004.13907 [cs.DC] A Rutgers Department of Computer Science Technical Report DCS-TR-750.

[116] Mohammadreza Soltaniyeh, Richard P. Martin, and Santosh Nagarakatte. 2021. SPOTS: An Accelerator for Sparse Convolutional Networks Leveraging Systolic General Matrix-Matrix Multiplication. arXiv:2107.13386 [cs.AR]

[117] Mohammadreza Soltaniyeh, Richard P. Martin, and Santosh Nagarakatte. 2022. An Accelerator for Sparse Convolutional Neural Networks Leveraging Systolic General Matrix-Matrix Multiplication. *ACM Trans. Archit. Code Optim.* (apr 2022). https://doi.org/10.1145/3532863 Just Accepted.

[118] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. 2021. Serpens: A High Bandwidth Memory Based Accelerator for General-Purpose Sparse Matrix-Vector Multiplication. https://doi.org/10.48550/ARXIV.2111.12555

[119] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3490422.3502357

[120] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang. 2020. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In *Proceeding of the Annual IEEE/ACM International Symposium on Microarchitecture*. 766–780. https://doi.org/10.1109/MICRO50266.2020.00068

[121] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. 2007. FreePDK: An Open-Source Variation-Aware Design Kit. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*. 173–174. https://doi.org/10.1109/MSE.2007.44

[122] P. D. Sulatycke and K. Ghose. 1998. Caching-efficient multithreaded fast multiplication of sparse matrices. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. 117–123. https://doi.org/10.1109/IPPS.1998.669899

[123] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* 8, 11 (jul 2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[124] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2020. How to Evaluate Deep Neural Network Processors: TOPS/W (Alone) Considered Harmful. *IEEE Solid-State Circuits Magazine* 12, 3 (2020), 28–41. https://doi.org/10.1109/MSSC.2020.3002140

[125] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9. https://doi.org/10.1109/CVPR.2015.7298594

[126] Urmish Thakker, Paul Whatmough, ZHIGANG LIU, Matthew Mattina, and Jesse Beu. 2021. Doping: A technique for Extreme Compression of LSTM Models using Sparse Structured Additive Matrices. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 533–549. https://proceedings.mlsys.org/paper/2021/file/a3f390d88e4c41f2747bfa2f1b5f87db-Paper.pdf

[127] W.F. Tinney and J.W. Walker. 1967. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE* 55, 11 (1967), 1801–1809. https://doi.org/10.1109/PROC.1967.6011

[128] Yuhsiang M. Tsai, Terry Cojean, and Hartwig Anzt. 2020. Sparse Linear Algebra on AMD and NVIDIA GPUs—The Race is On. In *ISC High Performance*. Springer, Springer. https://doi.org/10.1007/978-3-030-50743-5_16

[129] Stylianos I. Venieris, Javier Fernandez-Marques, and Nicholas D. Lane. 2021. unzipFPGA: Enhancing FPGA-based CNN Engines with On-the-Fly Weights Generation. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 165–175. https://doi.org/10.1109/FCCM51124.2021.00027

[130] Richard Vuduc, James W Demmel, and Katherine A Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16 (jan 2005), 521–530. https://doi.org/10.1088/1742-6596/16/1/071

[131] Richard Wilson Vuduc and James W. Demmel. 2003. *Automatic Performance Tuning of Sparse Matrix Kernels*. Ph.D. Dissertation. AAI3121741.

[132] Dong Wang, Ke Xu, Jingning Guo, and Soheil Ghiasi. 2020. DSP-Efficient Hardware Acceleration of Convolutional Neural Network Inference on FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 4867–4880. https://doi.org/10.1109/TCAD.2020.2968023

[133] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. 2016. DeepBurning: Automatic generation of FPGA-based learning accelerators for the Neural Network family. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2897937.2898002

[134] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/3061639.3062207

[135] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 9 pages.

[136] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2019. Adaptive Sparse Matrix-Matrix Multiplication on the GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. 68–81. https://doi.org/10.1145/3293883.3295701

[137] Mingrui Wu, Bernhard Schölkopf, and Gökhan Bakir. 2006. A Direct Method for Building Sparse Kernel Learning Algorithms. *Journal of Machine Learning Research* 7, 21 (2006), 603–624. http://jmlr.org/papers/v7/wu06a.html

[138] Xiaoru Xie, Jun Lin, Zhongfeng Wang, and Jinghe Wei. 2021. An Efficient and Flexible Accelerator Design for Sparse Convolutional Neural Networks. *IEEE Transactions on Circuits and Systems I: Regular Papers* 68, 7 (2021), 2936–2949. https://doi.org/10.1109/TCSI.2021.3074300

[139] Rui Xu, Sheng Ma, Yaohua Wang, Xinhai Chen, and Yang Guo. 2021. Configurable Multi-Directional Systolic Array Architecture for Convolutional Neural Networks. *ACM Trans. Archit. Code Optim.* 18, 4, Article 42 (July 2021), 24 pages. https://doi.org/10.1145/3460776

[140] Carl Yang, Aydin Buluc, and John D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. arXiv:1803.08601 [cs.DC]

[141] Leonid Yavits and Ran Ginosar. 2017. Sparse Matrix Multiplication on CAM Based Accelerator. *CoRR* abs/1705.09937 (2017). arXiv:1705.09937

[142] Xin You, Hailong Yang, Zhongzhi Luan, Yi Liu, and Depei Qian. 2019. Performance Evaluation and Analysis of Linear Algebra Kernels in the Prototype Tianhe-3 Cluster. In *Supercomputing Frontiers*, David Abramson and Bronis R. de Supinski (Eds.). Springer International Publishing, Cham, 86–105.

[143] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2019. Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 11 (2019), 2072–2085. https://doi.org/10.1109/TCAD.2017.2785257

[144] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 687–701. https://doi.org/10.1145/3445814.3446702

[145] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783723

[146] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240801

[147] Y. Zhang, Y. H. Shalabi, R. Jain, K. K. Nagar, and J. D. Bakos. 2009. FPGA vs. GPU for sparse matrix vector multiply. In *2009 International Conference on Field-Programmable Technology*. 255–262. https://doi.org/10.1109/FPT.2009.5377620

[148] Z. Zhang, H. Wang, S. Han, and W. J. Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA47549.2020.00030

[149] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 15–28. https://doi.org/10.1109/MICRO.2018.00011

[150] Y. Zhou, M. Yang, C. Guo, J. Leng, Y. Liang, Q. Chen, M. Guo, and Y. Zhu. 2021. Characterizing and Demystifying the Implicit Convolution Algorithm on Commercial Matrix-Multiplication Accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, Los Alamitos, CA, USA, 214–225.

[151] Chaoyang Zhu, Kejie Huang, Shuyuan Yang, Ziqi Zhu, Hejia Zhang, and Haibin Shen. 2020. An Efficient Hardware Accelerator for Structured Sparse Convolutional Neural Networks on FPGAs. arXiv:2001.01955 [eess.SY]

[152] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti. 2013. Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6.

[153] Uri Zwick. 2002. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM (JACM)* 49, 3 (2002), 289–317.