METHODOLOGY AND ANALYSIS FOR EFFICIENT CUSTOM ARCHITECTURE DESIGN USING MACHINE LEARNING

A Dissertation
Presented to
The Academic Faculty

By

Ananda Samajdar

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

Georgia Institute of Technology

December 2021

METHODOLOGY AND ANALYSIS FOR EFFICIENT CUSTOM ARCHITECTURE DESIGN USING MACHINE LEARNING

Thesis committee:

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

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

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Vijay Janapa Reddi
John A Paulson School of Engineering and Applied Sciences
Harvard University

Dr. Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Date approved: October 26, 2021

”I would rather have questions that can’t be answered than answers that can’t be questioned”

Richard Feynmann

To baba, how I wish you were here to see this day. To Ma, bhai, and Ranjita whose sacrifices made this day a reality.

ACKNOWLEDGMENTS

The last five years have been a roller coaster ride with a string of many news. I am thankful for so many people and lucky co-incidences that I do not think I will be able to accurately articulate in this section. Nevertheless, I will try my best in the following paragraphs to express my gratitude to the numerous people who have touched my life, and made this journey truly memorable.

First I would like to thank my family. My father who devoted his entire life and made countless sacrifices so that I can obtain the best possible education. My mother, who to me is the human embodiment of strength. Never faltered to keep me motivated and looking forward, even in the darkest of time. My younger brother, who has been a constant source of moral support. My fiance, who never lost faith in me, and has worked equally hard and made sacrifices so that I can fulfil my dreams.

I consider myself to be fortunate to have Dr. Tushar Krishna as my advisor. Tushar has been a constant source of positive will power and motivation. Not only he is an excellent scientist and a highly effective researcher, he is also an excellent mentor, compassionate friend and an inspiring leader. I would like to thank him for being patient with me both personally and professionally and helping me grow as researcher. I hope one day I can pay forward to my mentees.

I also want to extend my deepest gratitude to my thesis committee members, Prof. Saibal Mukhopadhyay, Prof. Hyesoon Kim, Prof. Vivek Sarkar, Prof. Vijay Janapa Reddi. I am grateful for their suggestions and support for my work. Their encouragements and constructive feedback has helped me immensely to improve my thesis.

I would like to specially mention Prof. Moin Qureshi for his support and encouragement over the years. I remember reaching out to Prof. Qureshi even before starting my studies at Georgia Tech. Over the years discussions with him and his insighful advice has helped me a lot in navigating the grad life.

My research work has been greatly shaped by discussions and exchange of ideas among my brilliant collaborators. Dr. Paul Whatmough, Dr. Yuhao Zhu, and Matt Mattina have motivated and supported me with development and increasing the reach of SCALE-Sim. In many ways, I owe it to Paul and Yuhao for showing me the direction in thinking about system design with custom architectures. Dr. Micheal Pellauer, Dr. Anshuman Parashar, Prof. Joel Emer are not only shining stars in the Machine learning accelerator design community, whose work greatly shaped my thought and understanding, but also have been great mentors who helped me multiple times along my PhD journey. I am thankful to Prof. Nachiket Kapre for helping me dip my hand in FPGA based acceleration. I am yet to meet another person who is so energetic and hands-on with actual development while also actively managing a research lab with multiple students. I am grateful to Mrs. Amrita Mathuriya and Dr. Sasi Manipatruni for having me over at Kepler and providing me with the glimpse of the entrepreneur life. I had the unique experience of working at their home during the early phase of Kepler. Thank you guys for treating me as an integral part of the Kepler family.

I have also had the good fortune to work with exceptionally brilliant student collaborators. Jun (Dr. Hyoukjun Kown) has practically mentored me to ramp up on the life of a PhD student. The Genesys project have been much harder to complete without the late night discussion and tremendous research inputs from Kartikay and Parth. Moritz (Dr. Jan Mortiz Joseph) is not only a brilliant scientist who I was fortunate to have as a team mate, but is also a very dear friend. I think I owe an apology to Eric to make him sit through my ranting sessions. You are one of the most patient people I know. Matthew and Geonhwa, thank you for helping me out so many times with my papers and endless discussions.

Thank you Divya, Guru, Poulami, Moumita, Prithayan. You guys are my family in the US and responsible for so many happy memory away from home. Also a big thanks to former and current members of the Synergy lab, Hyoukjun, Saeed, Felix, Mayank, Vineet, Yehowshua, William, Geonhowa, Raveesh, Abhimanyu, Parth, Aniruddh, Matthew. You guys are some of the most wholesome people that I know of and made my our lab a second home.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Machine learning algorithms especially Deep Neural Networks (DNNs) have revolutionized the arena of computing in the last decade. DNNs along the with the computational advancements also bring an unprecedented appetite for compute and parallel processing. Computer architects have risen to challenge by creating novel custom architectures called accelerators. However, given the ongoing rapid advancements in algorithmic development accelerators architects are playing catch-up to churn out optimized designs each time new algorithmic changes are published. It is also worth noting that the accelerator design cycle is expensive. It requires multiple iteration of design space optimization and expert knowledge of both digital design as well as domain knowledge of the workload itself. It is therefore imperative to build scalable and flexible architectures which are adaptive to work well for a variety of workloads. Moreover, it is also important to develop relevant tools and design methodologies which lower the overheads incurred at design time such that subsequent design iterations are fast and sustainable.

This thesis takes a three pronged approach to address these problem and push the frontiers for DNN accelerator design process. First, the thesis present the description of a now popular cycle accurate DNN accelerator simulator. This simulator is built with the goal of obtaining detailed metrics as fast as possible. A detailed analytical model is also presented in this thesis which enables the designer to understand the interactions of the workload and architecture parameters. The information from the model can be directly used to prune the design search space to achieve faster convergence. Second, the thesis details a couple of flexible yet scalable DNN accelerator architectures. Finally, this thesis describes the use of machine learning to capture the design space of DNN accelerators and train a model to predict optimum configurations when queried with workload parameters and design constraints. The novelty of this piece of work is that it systematically lays out the formulation of traditional design optimization into a machine learning problem and

also describes the quality and components of a model which works well across various architecture design tasks.

CHAPTER 1
INTRODUCTION

The pervasiveness of computing into our daily lives is irrefutably immense. It was only a couple decades ago when a single computer would typically serve the needs of a household. These days not only every person generally has a personal computers, but have smartphones, smart watches and other devices, which have computing power usually rivaling the performance of the aforementioned PCs. The breakneck advancements of computing has changed the way we are entertained, shop, and even commute in the most parts of the world. Among the many significant strides that moved the world of computing as we know it, some innovations stand out more than the others. In the last decade there have been two fundamental advances which has made tremendous impact on the direction of computer engineering research.

The first is the development of machine learning algorithms. Over the last few years, innovations in data science and machine learning has solved the problem like human level image perception and planning, natural language processing, data mining etc. which were thought to be impossible a few years ago. Deep neural networks are among the most noteworthy, which are capable of capturing complex representation spaces owing to their high dimensionality and learning capability using back-propagation algorithm. For the uninitiated, these networks can learn the mapping between two spaces given sufficient data proportional to the learnable parameters in the networks without any explicit programming.

While data is one attribute of enabling DNNs to achieve tremendous feat, yet another enabling factor contributing to the success is the rise of general purpose parallel processing on GPUs. The extensive parallelism and high computation demand of DNN training and inference provided a clear mandate to engineer more performant and parallel computing hardware. Traditionally computing relied over the voltage and frequency scaling properties of shrinking device sizes to extract both performance and efficiency among the generations of computers. However, at the beginning of the decade, the device sizes already reached sub 10nm level, which led to lower yields and other physical constraints which diminished the performance benefits the computer engineers traditionally depended on.

This brings us to the second significant advancement. As the hindrances encountered from device scaling became clear, computer architects turned to building custom architectures called accelerators to extract both performance and energy efficiency for the DNN algorithms. The need for custom design lead to a Cambrian explosion in custom designed DNN accelerator proposal from both academia or industry. For custom architecture design, tight integration with the workload is the key to achieve highly performant yet efficient designs. However, this also becomes a problem when the nature of the workload changes. Given the infancy of the field, there exist several challenges to ensure that the designs process is systematic and effective to generate performant as well as energy efficient instances of DNN acceleration. Moreover, advances in the machine learning community continue to build workloads that require more computation, have novel structures, and demand significantly different optimization goals between generations. *In this thesis, I study the design space of custom DNN accelerators and describe architectures, tools, models, and methodology to systematically design flexible yet scalable accelerators at the face of evolving AI workloads.* The next few sections describe the contributions and the structure of this thesis.

1.1 Thesis Contribution

In this thesis we address the challenges of systematically studying the design space of DNN accelerators and propose solutions to build flexible and scalable architectures which can accommodate the new and upcoming networks with diverse computing patterns and large computation demands. Figure 1.1 depicts the trifecta of directions that constitute the

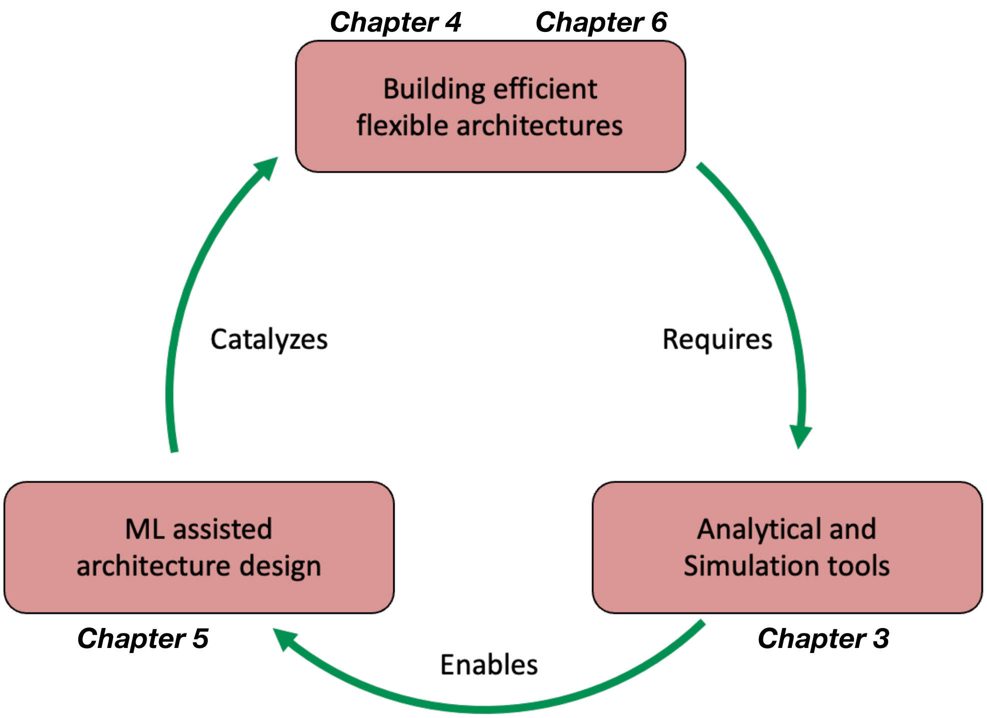

Chapter 4
Chapter 6
Building efficient flexible architectures
Catalyzes
Requires
ML assisted architecture design
Analytical and Simulation tools
Chapter 5
Chapter 3
Enables

Figure 1.1: The three main directions of research which build up this thesis

content of the thesis. As custom architectures become the norm to extract the performance as well as high efficiency, better exploration tools are required which can be used for rapid design optimization at minimum possible cost across the different workloads and implementation constraints. With access to fast and reliable tools, one can generate large amount of design space exploration data, mapping different implementation use cases and workloads to optimal design points. As we gather this data, we can employ the latest advancements in the machine learning algorithm to learn the optimization space of custom architecture and predict the optimal design points for future implementation or exploration tasks. As the figure depicts, Chapter 5 describes the contributions in this domain. This set of learnt model in turn, make the design space exploration task even faster and cheaper leading to invention of architectures which are capable of achieving even higher efficiencies. Chapter 4 and Chapter 6 describe two instances of accelerator design which are influenced by the tools developed as a part of this work. The following subsections provides a brief overview of the various solutions that this thesis proposes.

1.1.1 Systematic design decisions using analytical modeling and simulation

Chapter 2 describes the large variety of DNN accelerator architectures that have been proposed over the years. The various proposals touch and optimize of various aspects of the architecture, like interconnect, memory hierarchy, dataflows, the structure of the compute array and many other factors. While each of these components are important in themselves, the success of overall architecture design depends on the synergistic design of the individual components and the fit of the architecture and mapping strategy with the workloads.

In Chapter 3, I chose the simplest possible structure of systolic arrays and create an analytical model to capture the interaction of the workload parameters with the architecture parameters. The analytical model helps in optimizing the array design with respect to the workloads dimensions. The chapter also describes SCALE-Sim, a systolic array based simulator which provides cycle accurate compute and memory access to generate performance and efficiency metrics. The tool also provides metrics on interface behavior of the simulated design on a system level.

1.1.2 Scalable and flexible DNN accelerator design

In this thesis I also present two designs which demonstrate building building flexible and scalable accelerator design using the design principles laid out in Chapter 3. In Chapter 4, I describe a flexible accelerator implementation on Xilinx FPGAs. The presented design is different from existing FPGA implementation of DNN accelerators in several ways. First, the accelerator design is composed of building blocks implemented using existing DSP slices present in Xilinx FPGAs, which can be configured to work optimally for dense matrix-matrix and matrix-vector operations. Unlike the traditional practice of using LUTs in the configuration fabric to implement computation units, using preexisting hard DSPs enables faster clock speeds and hence higher performance. Second the design exploits the hardware cascades present in Xilinx FPGAs to provide configurable but dedicated links among the DSPs to scale the compute capability at runtime while maintaining ASIC like performance owing to the hard-wiring in the cascades.

The design presented Chapter 6, demonstrates a case for flexible and scalable architectures for ASIC based DNN acceleration. The proposed architecture leverages the leanings from Chapter 3 and depicts a reconfigurable systolic array based design, which can be morphed into a collection of distributed arrays or a monolithic array depending upon the fit for a workload.

1.1.3 Learning the accelerator design space using machine learning (ML)

This thesis presents a comprehensive methodology and example of leveraging machine learning for architecture design. To the best of my knowledge this thesis is the first to propose and demonstrate learning the design space of custom architectures. Traditionally, computer architecture focused on developing performant general purpose computer which are programmable for a large variety of workloads. Finding optimal design points, although a data driven decision making process, is conventionally manual and is dependent on iterative simulation and evaluation of the various points in the design space. With the advent of custom architectures, conventional techniques are no longer sufficient to find the optimal configurations in a cost effective way since the solution space has become diverse, complex, and in practical terms intractable to be effectively searched.

Figure 1.2 depicts a possible use-case for deploying learnt model for architecture optimization replacing search. As the figure depicts, there are two phases required to enable a learnt model based system to work which we call, *"Development"* and *"Production"*. The deployment starts at the development stage, where a model is constructed and trained on the optimal architecture and mapping prediction task using previously generated simulation data, or data obtained from actual implementations. Once the training converges,

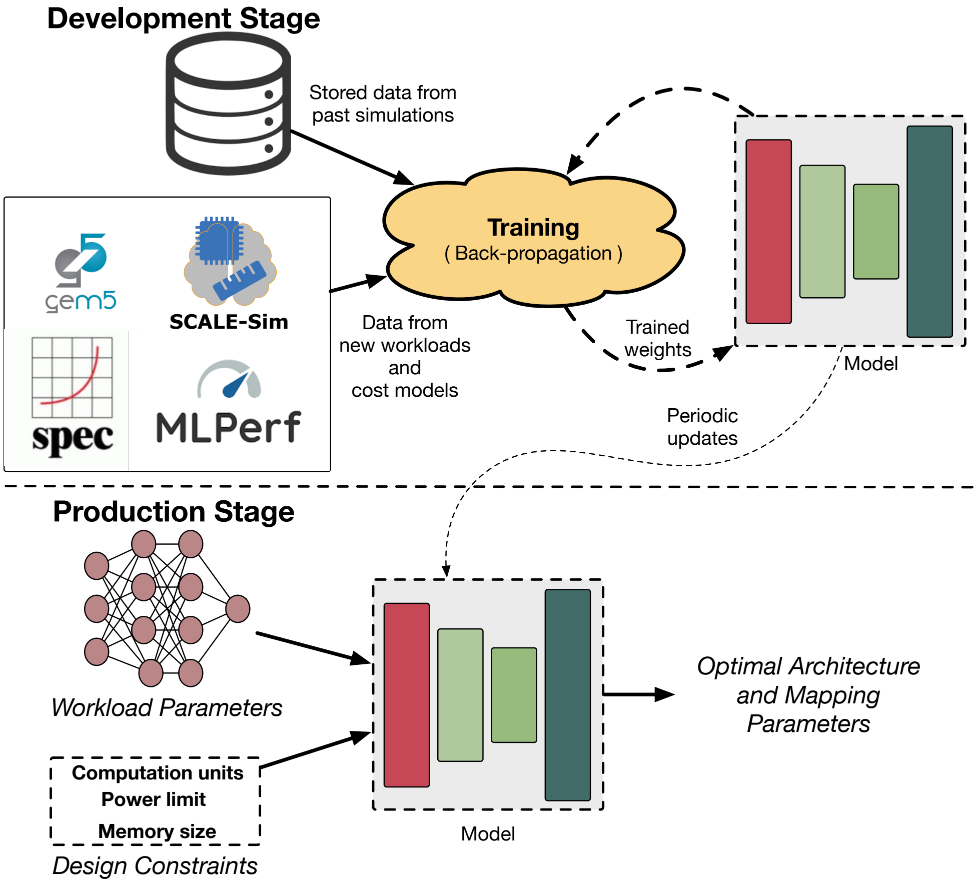


Figure 1.2: Schematic of a reference implementation scenario for deployment of a learnt model for aiding optimization of architecture and mapping parameters.

the learnt model is then deployed onto the production stage. It is in this stage where the model helps architecture designers with the optimization queries. As Figure 1.2 shows, in the production use case, the uses query the trained model with workload parameters, design constraints, and additional information about the system. The model then generates with the optimal parameters by performing a single inference. Given a single machine learning model inference is much faster than iterative search, the time and logistical of design space exploration is significantly reduced. While the production model serves queries from the customers, the development model is continuously updated with new training data to ensure that the distribution of design objectives learnt by the model is expanded regularly. The learnt weights from the development stage are used to periodically update the production model to keep it abreast with the changing design landscape.

Chapter 5 of the thesis details the formulation of the design space exploration problem into a ML problem, describes model creation and dataset generation to facilitate learning. Furthermore, in this thesis I perform both design aware, and design agnostic statistical analysis to get insights into the optimization spaces of various architecture design tasks. Furthermore, in Chapter 6, I show a specific use-case of learning the design and mapping space of a reconfigurable accelerator. The chapter describes AdaptNet, which is leaned neural network that can predict the optimal configuration and mapping to be used for a specific workload during execution on our design. The high accuracy of prediction, can be faithfully used to bypass caching, or highly cost intensive online searching techniques. In this chapter, I also advocate that due to the presence of the learned model, new class of reconfigurable architectures can now be deployed by coupling a highly flexible substrate with a configuration recommender. This thesis proposes the name *Self Adaptive Reconfigurable Arrays* or (SARA) for such designs, which were impossible to create without the learnt model simply because of the overheads of searching for optimal configurations in runtime.

1.2 Thesis Impact

Honors. The concept of using machine learning to capture the design space of DNN accelerators and hence using the learned model to obtain the optimal configuration by side stepping traditional search mechanism is generally well appreciated in the community. The initial proposal of AIrchitect won a silver medal at ACM student research competition (SRC) at ASPLOS 2019. Genesys[1] was also chosen as one of the finalists at ACM SRC at ASPLOS-2018.

Adoption and Follow on works. SCALE-Sim is a well received simulator used by many academic and industry research labs. SCALE-Sim's GitHub repository has about 225 stars and >100 forks. The simulator has been cited more than 130 times on Google Scholar since its release in 2018.

Tutorials. Three tutorial have been presented in top conferences (ISCA-2019, ASPLOS-2021, ISCA-2021) where I and my colleagues informed the community on using SCALE-Sim for their research and making contributions to the tools. The tutorials were attended by about 20 researchers in each iteration.

Book. The concepts of scaling and systematic design methodologies have been adopted in a couple of chapters in the synthesis lectures book titled, ”Data Orchestration in Deep Learning Accelerators”[2] published in 2020.

1.3 Thesis Statement

This thesis demonstrates the development of ML assisted self-adaptive hardware architectures for efficient execution of evolving AI workloads

1.4 Thesis Overview

As mentioned in the sections before, in this thesis, I discuss about the simulation and analytical infrastructure, instances and design principles for constructing flexible and scalable accelerators, and machine learning techniques to aid the design process of custom architectures. The rest of the thesis is organized as follows:

- Chapter 2 discusses the background information and the related works similar to the topics discussed in this thesis. This chapter is a summary of the existing literature.

- In Chapter 3, the thesis provides details on SCALE-Sim, which is a popular cycle accurate systolic array based DNN accelerator simulator. This chapter, also describes and analytical model to understand the interactions of architecture parameters of the compute array with the parameters of the workload for several mapping strategies. Finally the utility of both the analytical and simulation tool is demonstrated by conducting a study to systematically design DNN accelerators at scale.

- The next chapter, Chapter 4 demonstrates an instance of a DNN accelerator which is both flexible and scalable, implemented on Xilinx VU37P FPGA. The novelty of the proposed design lies in the fact that it purely uses the hardware cascades and interconnects already present in the FPGA to achieve high frequency operation, close to F_{max} of the FPGA. The chapter describes, the use to SCALE-Sim to determine the optimal mix of convolution and matrix-vector multiplication units for any given workload to get the maximum performance.

- Chapter 5, introduces the concept of machine learning assisted accelerator design. This chapter presents a detailed study on learning the accelerator design space using machine leaning and hence using the learned model to predict the optimal architectural parameters when queried with workloads and design constraints. Several novel concepts like formulating the traditional design optimization as ML problem, design aware and statistical analysis of the design datasets are presented in the chapter to provide thorough understanding of the design space and the possiblilty to learn it. The chapter also provides details about AIrchitect, a custom designed neural network recommendation model, which shows promising performance of learning the datasets and predicting the optimal design points for the case studies presented in this chapter.

- In Chapter 6, we consolidate the knowledge from the works presented in the previous chapters and demonstrate a flexible yet scalable reconfigurable accelerator can be constructed. The reconfigurable accelerator also incorporates a learnt machine learning model to find the optimal configuration of the accelerator at runtime for a given workload without taking the assistance of the software stack to search through the large configuration space. The leanrt model and the flexible accelerator are presented together as an instance of a new class of accelerators called Self Adaptive Reconfigurable Accelerators or SARA.

CHAPTER 2

BACKGROUND AND RELATED WORKS

2.1 Background

2.1.1 Computation in Deep Neural Network

Neural network and Multi layer perceptron. An artificial neural network is a set of mathematical operations which are inspired by the interconnection of physical neurons in a biological organism. Figure 2.1 depicts the logical structure of the artificial neuron and contrasts it with the biological counterpart. The biological neuron depicted in Figure 2.1(a) takes input signals from adjacent neurons through the dendrites. These signals in most cases are electrical pulses, which are amplified or attenuated by ion channels present in the connections between the neurons called synapses. The neuron body accumulates the input signals and generates an output electrical pulse based on specific firing mechanisms. The output signal is sent out to neighbouring neurons through the axon.

Akin to the biological counterpart, the basic building block in an artificial neural network is an artificial neuron which is depicted in Figure 2.1(b). The artificial neuron accepts the input vector which is weighted by each element of a weight vector. The weighted elements are first accumulated, and then passed via an activation function to generate the final output element.

Neural networks are constructed by grouping several neurons together in layers and then connecting the layers one after another. One of the simplest possible neural networks is the Multi-Layer Perceptron (MLP). Figure 2.2(a) shows a simple MLP neural network comprising of an input layer, and an output layer. Mathematically, the operations performed in each layer is equivalent to a matrix vector multiplication. In this operation the output vector is generated by multiplying the operand matrix with weights with the vector of

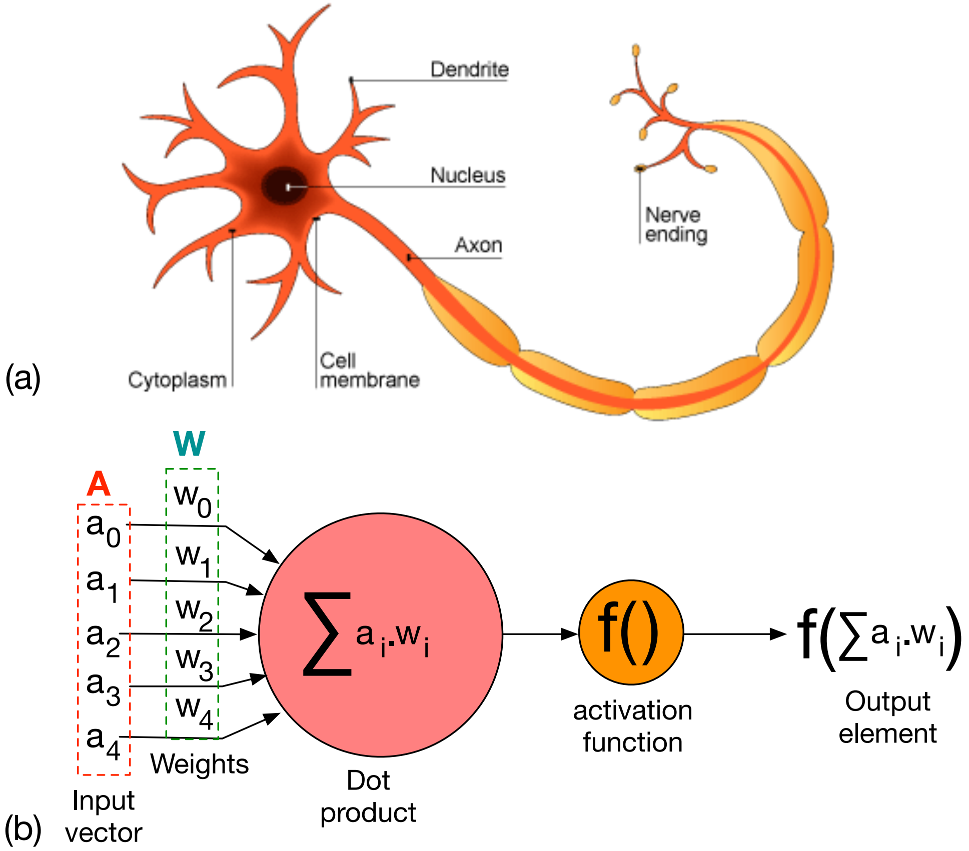


Figure 2.1: (a)Schematic of biological neural adapted from [3], (b) Logical structure of an artificial neuron

activations obtained from either the input or the output of the previous layer.

Convolution networks. Convolution neural networks are popular class of networks which comprise of special Conv2D layers followed by fully connected layers found in MLPs. The Conv2D layers work by multiplying the elements of the various filters with elements from only a portion of the input matrices in a sliding window pattern. The small portion of the input matrix used at a time is called a receptive field, which is inspired by the biological counterpart of the vision processing neural systems found in animals.

Figure 2.3(a) depicts an example convolution neural network and convolution layer. Mathematically the operation in a Conv2D layer is equivalent to a matrix-matrix multiplication operation. The operand matrices involved in the operations are (a) a matrix obtained by performing a *Im2Col* transformation of the input matrix, and (b) a matrix formed by

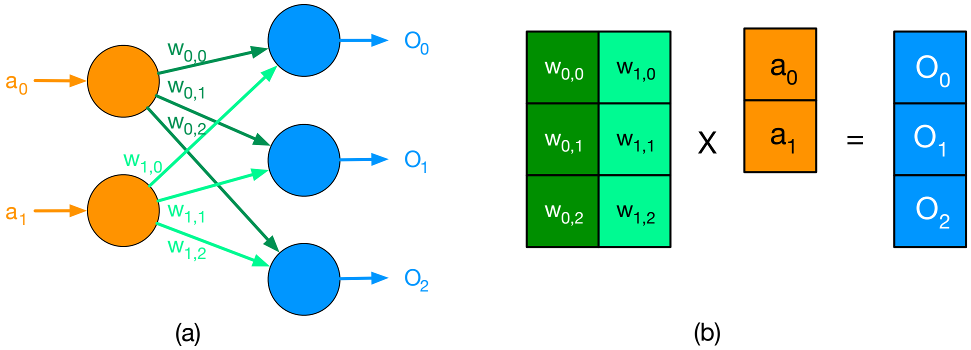


Figure 2.2: (a) Example of a simple MLP network with inputs, outputs, and weights (b) Equivalent matrix-vector multiplication

unrolling the different filter matrices and concatenating them along the rows. Figure 2.3(b) shows an example operand matrices obtained by *Im2Col* transformation of the input, and the weight matrix.

Recurrent network computation. Recurrent neural networks (RNNs) are another type of networks which bring in the notion of memory in to the networks. The salient feature in these networks is that unlike MLP or convolution networks the output generated by subsequent layers are used as inputs to the previous layers. Mathematically however, matrix-vector and matrix-matrix multiplications are equivalent to the operations performed in an RNN. In this thesis we primarily focus on accelerating MLPs and CNNs instead of RNNS.

2.2 Related Works

2.2.1 DNN Accelerator proposals

- **Early Accelerator Proposals.** The massive parallelism and relatively simple computation is a lucrative candidate for custom architecture implementation. Diannao [4] is one of the first hardware accelerator implementation, which not only focused on constructing efficient hardware structures but also presents a primitive memory hierarchy for the accelerator to extract the reuse of operands for CNNs and MLP networks. ShiDiannao [5], demonstrates that a new class of smart sensors can be

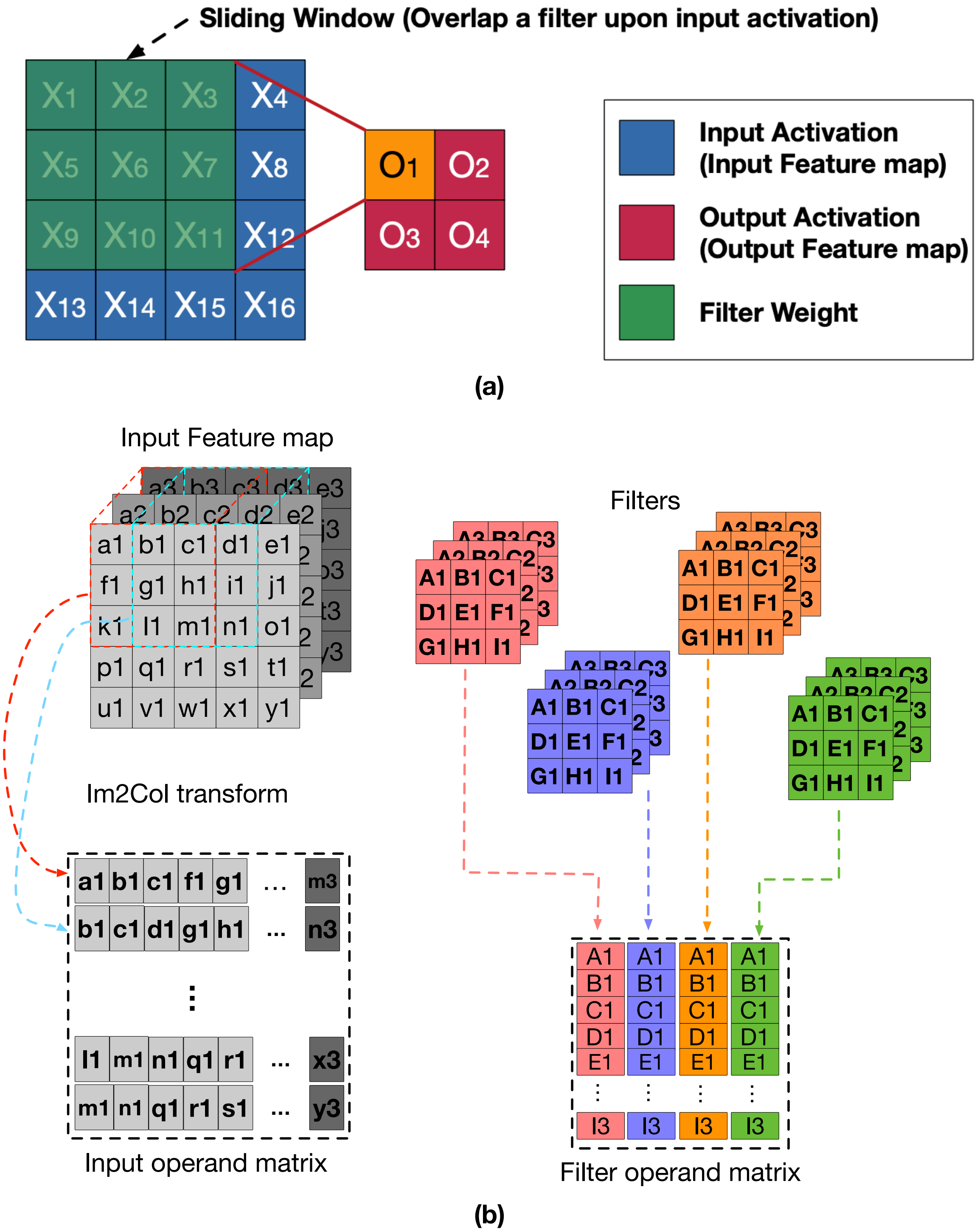


Figure 2.3: (a) Example of a simple convolution layer in a CNN (b) Formation of operand for equivalent matrix-matrix multiplications in a Conv2D layer using Im2Col transformation

designed by creating highly power efficient accelerators that can work directly on the sensor data, bypassing the system memory hierarchy. Eyeriss [6] is one of the seminal papers, which first systematically categorize the various dataflows that are used in DNN acceleration and their respective roles in improving the reuse and consequently the energy efficiency of the accelerator execution. This paper also introduces the row stationary dataflow, which is one of the first demonstrations of software-hardware co-design to improve the energy efficiency in DNN accelerators. Google's TPU [7] is a well known accelerator design which is one of the first ones to be coming from the industry. This large systolic array based accelerator architecture proposals first discussed the nature of inference workloads encountered at a data-center scale at the time and their implications in driving the design decisions. While the TPU opted for a throughput optimized design, Brainwave [8] from Microsoft demonstrated a latency optimized design. The Brainwave system is constructed by implementing accelerator instances over multiple instances connected via data-center networks to build a scaled-out system contributing to a single inference.

- **Flexible Accelerators.** The high diversity of workload dimensions lead to a huge divergence of optimal mapping and compute structures. This has led to an justifiable push towards design which can reconfigure or support multiple mapping strategies. MAERI [9] is one of the early designs which employ a highly configurable interconnect to create logical or virtual neurons, which is mapped onto a multiplier array to accommodate various workload dimensions. This design also implemented a adder tree structure which can emulate adder trees of different depths. While MAERI employed flexible substrate to accommodate for varying compute dimensions, Flexflow [10] proposed hardware structures to enable several mapping strategies. Zhang et al [11] on the other hand, argue that the inherent flexibility of FPGAs can be used to run designs tailored for individual networks rather than compromising on efficiency due to rigidity of ASIC based architectures. In their paper the authors propose an analytical optimizer, which generates the best micro-architecture for CNN and MLP workloads. Fused Layer CNN Accelerator [12] is another example of flexible accelerator design, which also uses FPGA based implementation. However, the authors employ a layer fusion strategy to extract the maximum reuse out of the input operands and generated activations to minimize off-chip accesses.

- **Accelerators Exploiting Sparsity.** A curious artifact of CNN and other DNNs using ReLU activation is that a significant amount of inputs, weights and generated activation values are either zero or close to zero. Furthermore, as the number of parameters grew in size regularization techniques lead to improved performance of the network but also inserts more zero valued activations leading to sparse operand matrices. Building accelerators which can bypass the redundant computation using zero valued operands can significantly improve the energy efficiency and performance of inference. EIE[13] is one of the first works to design accelerator hardware with sparsity in mind. This design, implemented in FPGA, identified the sparsity in the input activations and scheduled only those computations with non-zero operands. SCNN [14] from Nvidia is a well known design which takes a two step approach to exploit the advantages of sparse operands. The authors proposed to first construct tiles by parsing the sparse operand matrices and hence use a dense accelerator substrate with tracking structures to perform the computation for non-zero operands. CambriconX [15], is one of the first sparse matrix-multiplication based DNN accelerator which uses indexing structures in the hardware to remove redundant computation. A follow on work by the same team, Cambricon-S [16] proposes improvements in software to reduce the irregularity of the generated sparse activations and hence call for reducing complexity of the hardware indexing unit in the new accelerator architecture. SIGMA [17], on the other hand is a recent proposal which uses highly specialized interconnects to map sparse computation with irregular operand matrix sizes at a large scale.

- **Acceleration at scale** As the advances in machine learning research continue, more sophisticated model with increasing number of parameters are being developed and deployed in the recent years. Conventional accelerators struggle to provide adequate hardware support for such workloads which has triggered requirements for new designs which can perform efficient computation at scale. DaDiannao [18] is perhaps the first proposal which proposes a system comprised of interconnected accelerators to target large network training and inference at scale. Simba [19] is a recent scaled-out proposal from Nvidia, which demonstrate creation of an on-chip scaled out processor, composed of DNN accelerators connected together using a silicon interposer. Tangram [20] is a proposal from Stanford which also used Eyeriss like PEs to construct a larger accelerator connected by a new interconnect structures. Several new companies have recently been active in the building accelerators at scale. The most notable among them is probably Cerebras with their wafer scale processor [21]. Groq's tensor streaming processor[22] example of monolithic accelerator working at scale.

2.2.2 Simulation and Analytical Infrastructure for DNN accelerator Design

Over the course of past few years, there have been several simulation and analytical model proposals which complement SCALE-Sim with analytical models and simulation infrastructure, operating at various levels of details. Aladdin[23], is a tool which helps estimate power, performance and silicon area of arbitrary accelerators. The tool uses a HLS-like methodology and uses the C-code of the workload to estimate the regions of acceleration and hence the overhead of a potential accelerator design. Gem5-Aladdin[24] integrates the tool with the popular CPU simulator GEM-5[25]. MAESTRO[26], is an analytical cost model that provides pragmas to precisely describe the mapping of layers of DNN workloads and uses this information to infer the accelerator architecture and hence the various costs incurred. Marvel[27], extends MAESTRO's primitives to find the optimal scheduling strategy such that the off chip data movement cost is minimized. dMazerunner[28] is yet another tool, which uses the loop nest representation of the mapping space of DNN workloads to obtain the best mapping strategy by using clever techniques to prune the search space. Interstellar[29], is a similar tool to obtain energy efficient mapping of DNN workloads by analyzing Halide's [30] scheduling language. STONNE[31] is an recent full functional cycle level simulator infrastructure which models MAERI[9] and SIGMA[17] and can provide details of all execution steps of the workloads described in PyTorch[32].

2.2.3 Machine learning for assisting system design

Finding the optimal system design parameters is a data driven process. Furthermore, the size and complexity of design space is increasing at a pace such that traditional search based methods are no longer practical to find the optimal design parameters in a cheap and cost bounded manner. As a consequence, researchers have been exploring using machine learning techniques to improve the quality and convergence time to search for the optima. The following paragraphs list a few recent works.

- **ML for Architecture search:** Apollo [33] is a recent work from Google, targeting sample efficient searching through the accelerator design space using reinforcement learning. Gamma[34] and ConfuciuX[35] are similar ML based architecture mapping and design space configuration search methods which use genetic algorithm and reinforcement learning (RL) respectively. AutoTVM[36] use ML model for cost prediction to improve fast mapping search during compile time.

- **ML for EDA:** Recently there has been a significant push toward automating place-and-route using machine learning. Mirhoseni et al[37] use RL for task placement on a heterogeneous system. Wang et al[38] use GCN and RL for automatic transistor sizing. NVCell[39] is a RL based proposal from Nvidia to automate standard cell placement. Nautilus[40] uses genetic algorithm to improve FPGA place and route. Kwon et al[41], use online tensor-based recommender systems to aid place and route in chip design.

CHAPTER 3

ANALYTICAL MODELING AND SIMULATION INFRASTRUCTURE FOR SYSTEMATIC CHARACTERIZATION AND DNN ACCELERATORS

3.1 Introduction

DNN accelerators extract efficiency and performance simultaneously by employing customized optimization for target workloads and employing innovative structure which are not available on a general purpose computing chip. The cost of this customization however is the risk of obsolescence when the nature of the workload changes significantly.

Given the ever changing landscape of the deep learning and machine learning in general, the risk of such obsolescence is often significant. Finding new designs quickly while minimizing cost is therefore a top requirement and requires in depth knowledge of both the workload and its interplay with the architecture and mapping parameters. Any infrastructure that can provided detailed information about the performance of various designs within the changing landscape of workloads is therefore extremely valuable to systematically explore the design space and find optimal implementations.

In this chapter, two such tools are presented. First the chapter details SCALE-Sim, which is a cycle accurate simulation infrastructure for systolic array based DNN accelerators (Section 3.2.3). SCALE-Sim is designed to provide cycle level details on both on-chip and off-chip transactions of operand and output elements, and also generates metrics on performance, mapping, and efficiency much faster that traditional RTL simulation. This chapter also presents the work performed by me and my colleagues on designing an analytical framework to understand the performance and mapping of GEMM workloads on a systolic array based accelerator (Section 3.3). The merit of the analytical model is that, the closed for equations help identify the most important architectural parameters

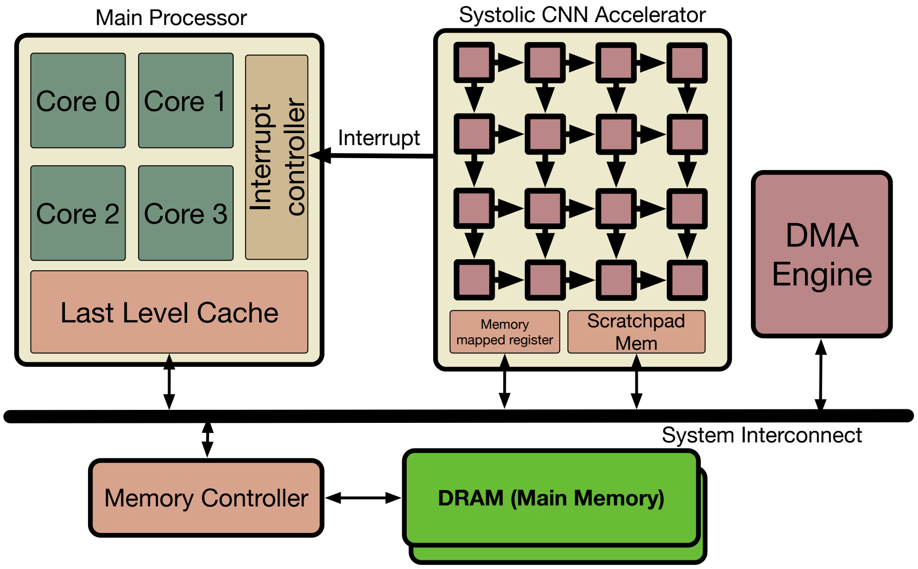


Figure 3.1: Schematic showing the integration model of accelerator in a systems context

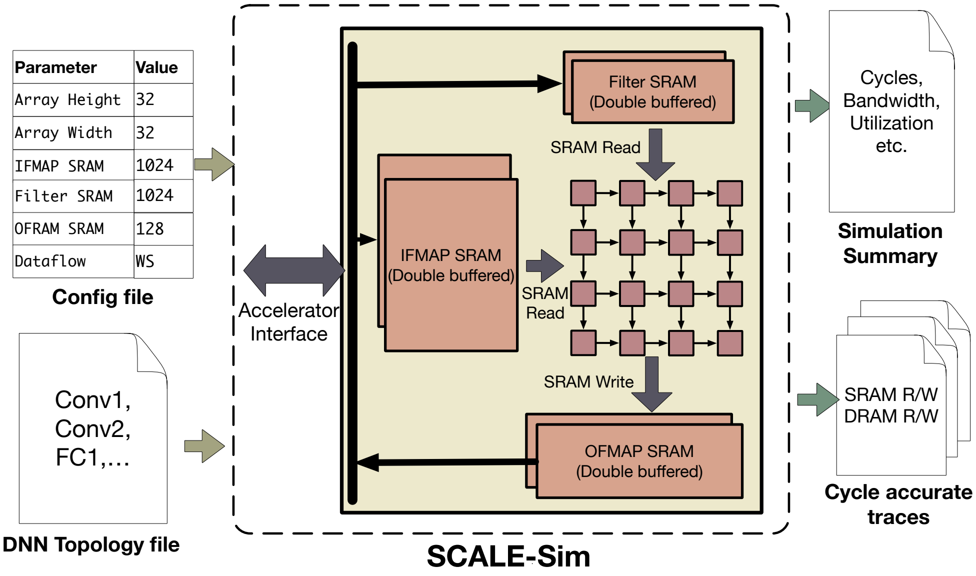
that need to be optimized prior to running a simulation. This helps pruning the space for design space exploration and help in faster convergence. Next in Section 3.4, we show the utility and effectiveness of both the tools in a case study to identify strategies for scaling DNN accelerators.

3.2 SCALE-Sim: Systolic Accelerator Simulator

SCALE-Sim is a cycle-accurate behavioural simulator that provides a publicly available open-source modeling infrastructure for array-based DNN accelerators. SCALE-Sim enables designers to quickly iterate over and validate their upcoming designs with respect to the various optimization goals for their respective implementation points. In this section, we first provide some background on systolic arrays and second, we describe our modeling methodology.

3.2.1 Background: Systolic Arrays and Dataflows

Systolic arrays are a class of simple, elegant and energy-efficient architectures for accelerating general matrix multiplication (GEMM) operations in hardware. They appear in many



Parameter	Value
Array Height	32
Array Width	32
IFMAP SRAM	1024
Filter SRAM	1024
OFRAM SRAM	128
Dataflow	WS

Figure 3.2: Schematic depicting the inputs needed and the outputs generated by SCALE-Sim

academic and commercial DNN accelerator designs [42, 43, 44]. An overview of system integration is shown in Figure 3.1

Compute. The compute microarchitecture comprises several Multiply-and-Accumulate (MAC) units (also known as Processing Elements, or PEs), connected in a tightly coupled two dimensional mesh. Data is fed from the edges from SRAMs, which then propagates to the elements within the same row (column) via unidirectional neighbour-to-neighbour links. Each MAC unit stores the incoming data in the current cycle in an internal register and then forwards the same data to the outgoing link in the next cycle. This store and forward behavior results in significant savings in SRAM read bandwidth and can very effectively exploit reuse opportunities provided by convolution/GEMM operations, making it a popular choice for accelerator design. Note that this data movement and operand reuse is achieved: (1) without generating or communicating any address data, and (2) only using hard-wired local register-to-register inter-PE links, without any interconnect logic or global wires. For these two reasons, the systolic array is extremely energy and area efficient.

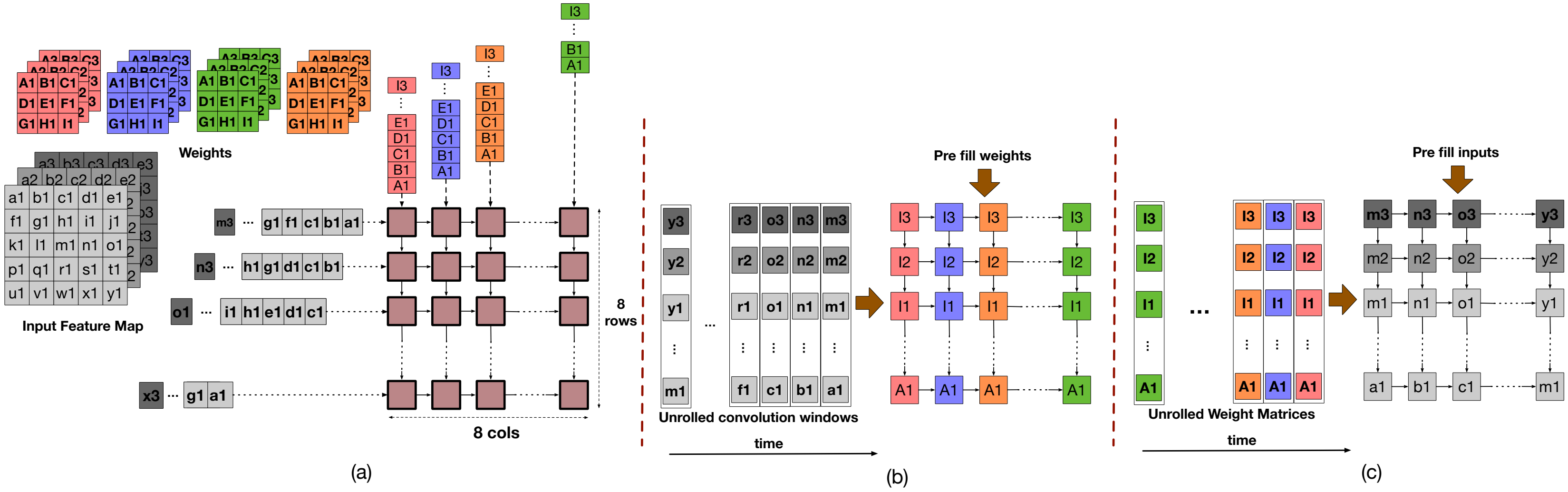


Figure 3.3: Schematic showing the mapping in various dataflows (a) Output stationary; (b) Weight stationary; (c) Input stationary

Memory. Systolic Arrays are typically fed by local linearly-addressed SRAMs on the two edges of the array, with outputs collected along a third edge. These local SRAMs are often double buffered and are backed by the next level of the memory hierarchy.

Data Reuse. A typical convolution can be viewed as a small filter kernel being slid over a given input matrix, with each overlap generating one output pixel. When the convolution operation is formulated as successive dot-product operations, three reuse patterns are immediately evident:

- Each convolution window uses the same filter matrix, to generate pixels corresponding to a given output channel.
- The adjacent convolution windows share portions of the input matrix if the stride is smaller than window dimension.
- To generate a output pixel in different output channels, different filter matrices use the same convolution window.

These reuses can be exploited via the dataflow or mapping of the DNN over the array.

Dataflow. There are three distinct strategies of mapping compute or *dataflows* onto the systolic array named *Output Stationary (OS)*, *Weight Stationary (WS)*, and *Input Stationary (IS)* [6] as shown in Figure 3.3. The "stationarity" of a given dataflow is determined by the tensor whose element is not moved (i.e. stationary) for the maximum duration of time throughout the computation. Although many different dataflows exist for spatial arrays, we only consider true systolic dataflows that only use local communication.

The OS dataflow depicted in Figure 3.3(a), therefore refers to the mapping where each MAC units is responsible for all the computations required for a OFMAP pixel. All the required operands are fed from the edges of the array, which are distributed to the MAC processing elements (PE) using internal links to the arrays. The partial sums are generated and reduced within each MAC unit. Once all the MAC units in the array complete the generation of output pixels assigned to itself, the peer to peer links are used to transfer the data out of the array. No computation takes place in the array during this movement. An alternative high performance implementation using a separate data plane to move generated output is also possible, however, it is costly to implement.

The WS dataflow on the other hand uses a different strategy as shown in Figure 3.3(b). The elements of the filter matrix are prefilled and stored into each PE prior to the start of computation, such that all the elements of a given filter are allocated along a column. The elements of the IFMAP matrix are then streamed in through the left edge of the array, and each PE generates one partial sum every cycle. The generated partial sums are then reduced across the rows, along each column in parallel to generated one OFMAP pixel (or reduced sum) per column.

The IS dataflow is similar to WS, with the difference being in the order of mapping. Instead of pre-filling the array with elements of the filter matrix, elements of the IFMAP matrix are stored in each PE, such that each column has the IFMAP elements needed to generate a given OFMAP pixel. Figure 3.3(c) depicts the mapping. We describe these dataflows in more detail in Section 3.3.2.

3.2.2 System Integration

We consider the typical offload model of accelerator integration in SCALE-Sim. We attach the DNN accelerator to the system interconnect, using a slave interface on the accelerator, as illustrated in Figure 3.1. The CPU is the bus master which interacts with the accelerator by writing task descriptors to memory-mapped registers inside the accelerator. When a task is offloaded to the accelerator, the CPU master can context switch to progress other jobs, while the accelerator wakes up and starts computing, independently generating its memory requests and side channel signals. When the computation has finished, the accelerator notifies the CPU, which accesses the results from the accelerator internal memory.

Thus, the cost on the system performance for integrating an accelerator is the extra accesses on the system bus, which could be modelled as interface bandwidth requirement. SCALE-Sim allows for modeling the main memory behavior by generating accurate read and write bandwidths at the interface, which can then be fed into a DRAM simulator e.g., DRAM-Sim2[45].

3.2.3 Implementation

Internally, SCALE-Sim takes an inside-out implementation approach. Specifically, the simulator assumes that the accelerator is always compute bound and the PEs are always used to the maximum possible utilization - as dictated by the dataflow in use. With this implementation model, the simulation in SCALE-Sim takes place in following steps.

- SCALE-Sim generates cycle accurate read addresses for elements required to be fed on the top and left edges of the array *such that the PE array never stalls*. These addresses are effectively the SRAM read traffic for filter and input matrices, as dictated by the dataflow. Given the reduction takes a deterministic number of cycles after the data has been fed in, SCALE-Sim generates an output trace for the output matrix, which essentially constitutes the SRAM write traffic.

- SCALE-Sim parses the generated traffic traces, to determine total runtime for compute and data transfer to and from SRAM. The data transfer time is essentially the cycle count of the last output trace entry. The SRAM trace also depicts the number of rows and columns that have valid mapping in each cycle. This information couples with the dataflow is used to determine the utilization of the array, every cycle.

- In SCALE-Sim the elements of both the input operand matrices, and the generated elements of the output matrix is serviced by dedicated SRAM buffers backed via a double buffered mechanism, as shown in Figure 3.2. As the sizes of these buffers are known from user inputs, SCALE-Sim parses the SRAM traces and determines the time available to fill these buffers such that no SRAM request is a miss. Using this

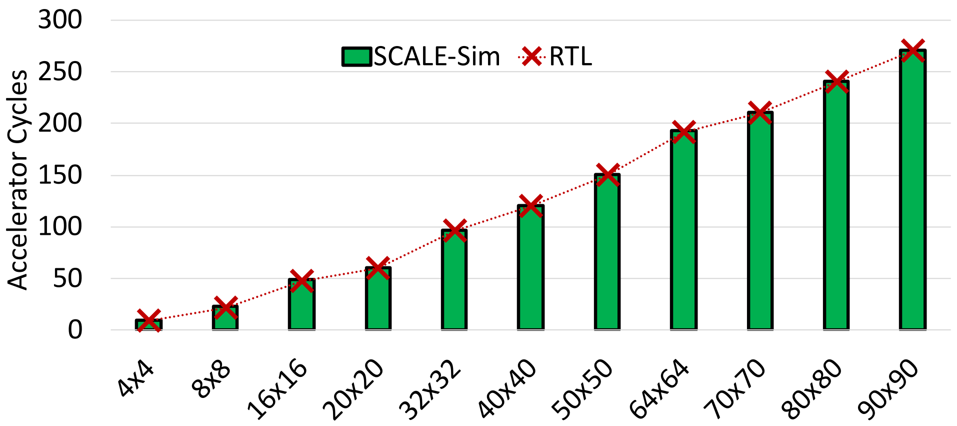


Figure 3.4: Figure depicting the cycles obtained by RTL implementation and SCALE-Sim simulation for varying array sizes under full utilization

interfaces SCALE-Sim generates a series of prefetch requests to SRAM which we call the DRAM trace.

- The DRAM traces are the used to estimate the interface bandwidth requirements for the given workload and the provided architecture configuration.
- The trace data generated at the SRAM and the interface level is further parsed to determine the total on-chip and off-chip requests, compute efficiency, and other high level metrics.

3.2.4 Validation of the tool

We validated SCALE-Sim against an RTL implementation of a systolic array. Figure 3.4 depicts the cycles obtained when matrix multiplications are performed on varying arrays sizes (X-axis) under full utilization with OS dataflow, from RTL implementation and SCALE-Sim simulations. As depicted by the figure the cycle counts obtained by both the methods are in good agreement.

Table 3.1: SCALE-Sim config description

Parameter	**Description**
ArrayHeight	Number of rows of the MAC systolic array
ArrayWidth	Number of columns of the MAC systolic array
IfmapSRAMSz	Size of the working set SRAM for IFMAP in KBytes
FilterSRAMSz	Size of the working set SRAM for filters in KBytes
OfmapSRAMSz	Size of the working set SRAM for OFMAP in KBytes
IfmapOffset	Offset to the generated addresses for IFMAP px
FilterOffset	Offset to the generated addresses for filter px
OfmapOffset	Offset to the generated addresses for OFMAP px
DataFlow	Dataflow for this run. Legal values are 'os','ws', and 'is'
Topology	Path to the topology file

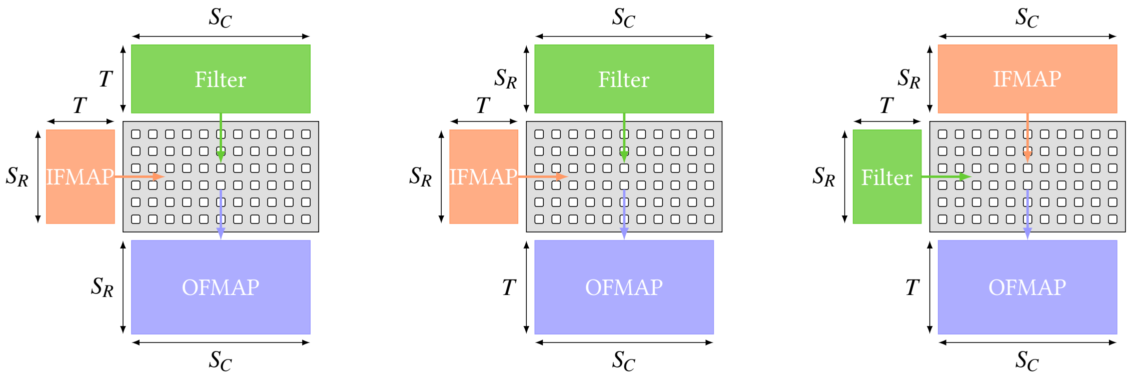
3.2.5 User Interface

Figure 3.2 depicts the inputs files used by the simulator, the outputs that are generated. SCALE-Sim takes two files as input from the user: one is a hardware configuration, and the other is a neural network topology for the workload. The configuration file contains the user specification for architectural parameters, like the array size, the memory size, and the path to the topology file. Table 3.1 depicts the complete list of parameters, which are mostly self-explanatory. For layers such as fully-connected (i.e. matrix-vector), the input parameters correspond to convolutions where the size of the filters are same as that of the IFMAP.

The topology file contains the layer topology dimensions for each of the layers in the given neural network workload. This is a comma-separated value (CSV) file, with each row listing all the required hyper-parameters for a given layer – Table 3.2 gives the complete list of all the entries in a given row. SCALE-Sim parses the topology file one line at a time and simulates the execution of the layer. This is a natural approach for traditional neural networks which are primarily composed of a single path. However, modern DNNs often contain "cells" that are composed of multiple convolution layers in parallel [46]. SCALE-Sim serializes the execution of such layers in the same order in which they are listed in the topology file.

Table 3.2: SCALE-Sim Topology file description

Parameter	Description
Layer Name	User defined tag
IFMAP Height	Dimension of IFMAP matrix
IFMAP Width	Dimension of IFMAP matrix
Filter Height	Dimension of one Filter matrix
Filter Width	Dimension of one Filter matrix
Channels	Number of Input channels
Num Filter	Number of Filter matrices. This is also the number of OFMAP channels
Strides	Strides in convolution



(a) Output stationary dataflow (b) Weight stationary dataflow (c) Input stationary dataflow

Figure 3.5: Data Flow Mapping

SCALE-Sim generates two types of outputs. First is the cycle accurate traces for SRAM and DRAM reads and writes. The traces are also CSV files, which list the cycle and the addresses of data transferred in a given cycle. The other type of output files are reports with aggregated metrics obtained by parsing information from the traces. These include cycle counts, utilization, bandwidth requirements, total data transfers etc. The trace-based methodology is very easy to debug and highly-extensible to new analyses and architectures.

3.3 Analytical model for runtime

In SCALE-Sim, all the simulated metrics including runtime are determined at the end of a round of simulation. However running simulation for all possible data points in a large search space is expensive and sometimes unnecessary. In this section we describe an

Table 3.3: Spatio-Temporal Allocation of DNN Dimensions

	Spatial Rows (S_R)	**Spatial Columns (S_C)**	**Temporal (T)**
Output Stationary	N_{ofmap}	N_{filter}	W_{conv}
Weight Stationary	W_{conv}	N_{filter}	N_{ofmap}
Input Stationary	W_{conv}	N_{ofmap}	N_{filter}

N_{filter} : Number of convolution filters
N_{ofmap}: Number of OFMAP pixels generated by filter
W_{conv} : Number of partial sums generated per output pixels

effective analytical model for runtime, which accounts for the data movement patterns simulated by SCALE-Sim. Please note however, the analytical model does not model the memory accesses and bandwidth demand arising due to limited memory which is captured by SCALE-Sim. We use this model to estimate costs and prune the search space for the subsequent scalability study described in Section 3.4.

3.3.1 Mapping across Space and Time

In dense DNN computations, running different types of layers generalize to matrix-matrix multiplications of different sizes. For systolic arrays, we consider the operand matrices of dimensions $S_R \times T$ and $T \times S_C$ respectively, where S_R and S_C are the spatial dimensions along which computation is mapped, and T is the corresponding temporal dimension. These matrices are obtained by projecting the original operand matrices into the available spatio-temporal dimensions. For example, for multiplying matrices of size $M \times K$ and $K \times N$, the dimension M is mapped to S_R, dimension N is mapped to S_C and the dimension K to T.

Figure 3.5 illustrates the mapping of a 2D convolution onto the three dataflows. Figure 3.5a shows the mapping corresponding to output stationary (OS) dataflow. The first operand matrix, with size $S_R \times T$, is a rearranged input feature map (IFMAP) matrix. Each row consists of elements corresponding to one convolution window, while the number of rows is the number of OFMAP pixels generated per filter. The second operand matrix contains unrolled filter elements, with each filter unrolled along each column, resulting in a $T \times S_C$ matrix.

Figure 3.5b and Figure 3.5c depict the mapping for other two dataflows; *Weight Stationary (WS)* and *Input Stationary (IS)*. For WS, the number of convolution windows maps to S_R, while S_C is equal to the number of filters. As seen in Section 3.2 the partial sums for each OFMAP pixel are generated every subsequent cycle making the mapping along the temporal dimension T equal to the number of OFMAP pixels generated. In the IS dataflow however, the order and direction of feeding the IFMAP matrix and the filter matrices are interchanged. This implies that the mapping along the S_R and S_C dimensions for this dataflow is the same size as the convolution window and number of OFMAP pixels generated per filters respectively. While the temporal dimension T maps the number of filters. Table 3.3 summarizes these dimensions.

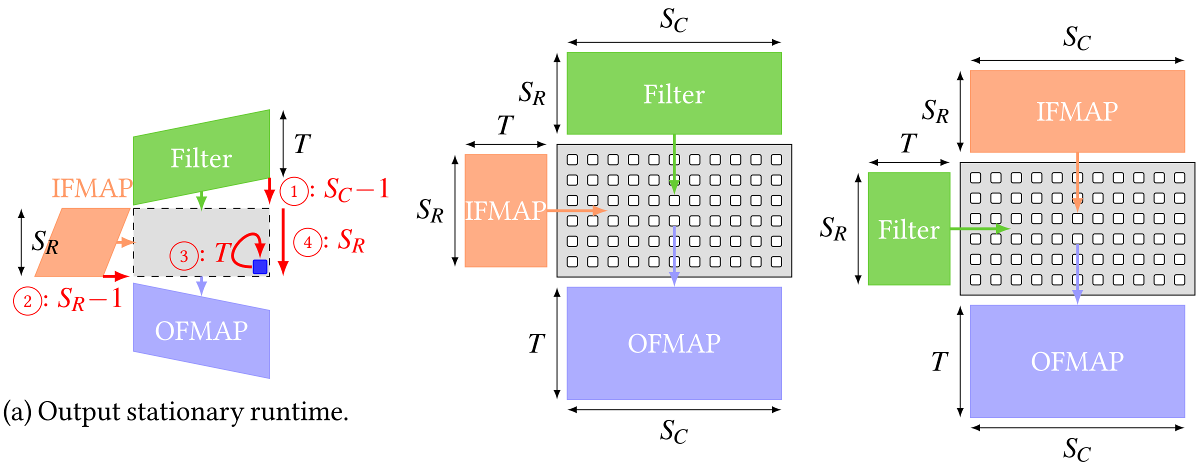
3.3.2 Runtime for Scale-Up

With the above abstraction of mapping in place, it is feasible to model the runtime for various dataflows, under the assumption of either a restricted or unrestricted number of compute elements. In our discussions we will only use multiply-and-accumulate (MAC) units as the compute elements within the systolic array.

Runtime with unlimited MAC units

Given an unlimited amount of MAC units, the fastest execution for any dataflow is achieved using the maximal array size of $S_R \times S_C$. However, note that even though all the multiplication operations are done in one cycle, the runtime needs to account for both the store and forward nature of the array, and the existence of the temporal dimension T (> 0).

Figure 3.6 shows the steps followed for moving data in the three dataflows introduced in Section 3.2. Figure 3.6a depicts the steps when implementing the OS dataflow. As mentioned before the IFMAP matrix is fed from the left while the filter elements are pushed in from the top edge. To account for the store and forward nature of the arrays and match



(a) Output stationary runtime.

(b) Weight stationary runtime.

(c) Input stationary runtime.

Figure 3.6: Schematic depicting steps to model runtime for dataflows in systolic array.

the data arrival time at all the PEs, the data distribution is skewed; the PE at the top left corner of the array receives both the operands at the first cycle, the PEs in the next column and next row get their operands in the next cycles, their neighbours in the cycle after that and so on. The PE at the bottom right corner of the array (marked in blue), is the last to receive the operand data. It is easy to see that the cycle at which the first operands arrive at this PE is $S_R + S_C - 2$ (adding steps ①, ② and ③). In this dataflow, each PE receives two operands per cycle and generates one OFMAP pixel value by in-place accumulation. It takes T cycles to generate on output, which is equal to the number of elements in a convolution window. The generated outputs are taken out from the bottom edge of the array. While it is possible to take out the output along other edges as well, using the bottom edge is the fastest alternative. The time required to completely drain the array of the generated output is S_R cycles after the PE at the right most corner has finished computation (step ④). Therefore, the total time taken for entire computation is,

$$\tau_{scaleup\_min} = 2S_R + S_C + T - 2 \tag{3.1}$$

In Figure 3.6b we perform the same analysis for WS dataflow. Here, the filter matrix is fed into the array from the top and is kept alive until the computations involving these operands are complete. Skewing is not needed as no computation is taking place while the filters are being fed. This takes S_R cycles (step ①). Once the filter elements are in place, the elements of the IFMAP matrix are fed from the left edge of the array. Each PE reads the IFMAP operand, multiplies it with the stored weight and forwards the partial sum to the PE in the neighbouring row for reduction. The first data arrives at the last row after $S_C - 1$ cycles (step ②). The IFMAP matrix is fed in one column at a time, therefore every column in the systolic array receives T operands, one each cycle, corresponding to the number of columns in the IFMAP matrix (step ③). Furthermore, for all the partial sums generated reduction occurs across the rows, for each column. After the top row receives and operand from the IFMAP, it takes $S_R - 1$ cycles to reduce (step ④). Therefore the array is drained out of all partial sums, after reduction happens in the rightmost column. The total runtime therefore is,

$$\tau_{scaleup\_min} = 2S_R + S_C + T - 2$$

Using similar analysis and Figure 3.5c, we can show that the above expression holds true for the IS dataflow as well. *Thus Equation 3.1 captures the runtime for all the dataflows in a systolic array when the number of MAC units is infinitely large*

Runtime with limited MAC units

Having a large enough systolic array which can map all the compute at once is often not practically feasible. Due to the large amount of computation compared to hardware compute units, it is necessary to tile the workload into chunks. We term this practice as *folding* where each of these chunks are called a *fold*[1]. Folds can be generated by slicing the compute along the S_R and S_C dimensions. When using a $R \times C$ array, the number of fold

[1] This is often also known as tiling

along rows (F_R) and columns (F_C) are determined as follows.

$$F_R = \lceil S_R/R \rceil \; F_C = \lceil S_C/C \rceil \tag{3.2}$$

Figure 3.7 illustrates this.

Analysis similar to Section 3.3.2 can be used to express the time taken in each of these folds as is given by the following equation, for all dataflows.

$$\tau_F = 2R + C + T - 2 \tag{3.3}$$

Where R and C are the rows and columns of the systolic array and T is the temporal dimensions. The total runtime can therefore be expressed from Equation 3.2 and Equation 3.3 as following.

$$\tau_{scaleup} = (2R + C + T - 2)\lceil S_R/R \rceil \lceil S_C/C \rceil \tag{3.4}$$

The above equation provides us with the insights on the factors affecting runtime. For a given workload and array configuration, choice of dataflow assigns the values for S_R, S_C and T respectively, which could be selected to minimize τ. On the other hand if the workload and dataflow is fixed, for a given number of MAC units, the optimal values of R and C could be determined to reduce the runtime as well.

Equation 3.4 can be used to determine the optimal configuration for a given matrix by implementing search over the possible R and C values. For workloads with multiple matrix operations, this model can be used as a cost model as depicted later in Section 3.4.2.

3.3.3 Optimal Partitioning for Scale-Out

In our previous analysis we have only considered a single array to study the affect of micro-architectural and design parameters on runtime. Instead of creating a single monolithic architecture with multiple PEs (i.e., scale-up), an alternative design choice is to employ

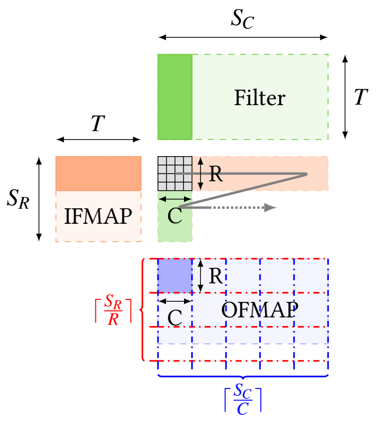


Figure 3.7: Scale Up

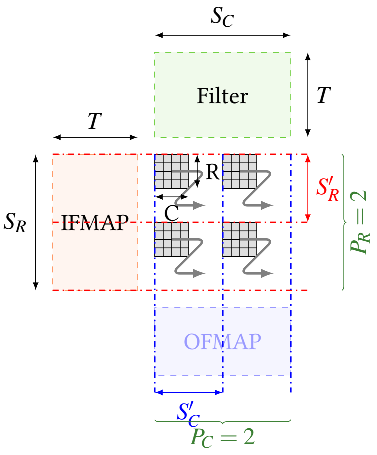


Figure 3.8: Scale Out

multiple units of systolic arrays, each responsible for one partition of the output feature map, to increase the available parallelism (i.e., scale-out) In this section we will model the runtime of such systems.

The scaled out configuration introduces another set of parameters, as shown in Figure 3.8. Unlike in scale-up where all the MAC units are arranged in a $R \times C$ array, in scaled-out configuration, the MAC PEs are grouped into $P_R \times P_C$ systolic arrays, each with a PE array of $R \times C$.

Using this approach for a given number of partitions $P = P_R \times P_C$, the effective workload mapped for computation over each partition can be determined by,

$$S'_R = \lceil S_R/P_R \rceil,\ S'_C = \lceil S_C/P_C \rceil \tag{3.5}$$

Within each array, we can use Equation 3.4 to decide the optimal aspect ratio ($R \times C$) for running the partitioned workload. Since the individual partitions execute in parallel, the total runtime of the scaled-out system is simply the runtime of the slowest cluster which can be determined by Equation 3.4 and Equation 3.5

$$\tau_{scaleout} = (2R + C + T - 2)\lceil S'_R/R \rceil \lceil S'_C/C \rceil \tag{3.6}$$

Table 3.4: Matrix dimensions of our language model workloads. mapped to S_R, S_C, and T

Name	S_R	T	S_C
GNMT0	128	4096	2048
GNMT1	320	4096	3072
GNMT2	1632	1024	36548
GNMT3	2048	32	4096
DB0	1024	50000	16
DB1	35	2560	4096
TF0	31999	84	1024
TF1	84	4096	1024
NCF0	2048	128	1
NCF1	256	2048	256

3.4 Analysis of Scaling

The primary aim of scaling a hardware accelerator, is to improve the runtime of a given workload. Since there are many ways of scaling a system, the first natural question to ask is whether any one of the methods proves beneficial over the others. To answer this question, we computed runtime using the analytical model described in Section 3.3, when using different configurations of monolithic vs scaled out arrays, given the same budget for MAC units. For workloads in our experiments, we used the convolution layers in Resnet50 CNN [46] and a few representative layers from widely used contemporary natural language processing models: GNMT[47], DeepSpeech2 [48], Transformer [49], and neural collaborative filtering [50]. The matrix dimensions corresponding to these workloads are detailed in Table 3.4

Search Space for Scale-up and Scale-out. Figure 3.9(a) provides the glimpse of the search space associated with the problem at hand. Each marker in the figure depicts a design point for corresponding to five different compute capabilities denoted by number of MAC units. On the x axis we have all possible dimensions for a systolic array with these mac units. The y axis represents the partitioned configurations when scaling out. We limit the smallest systolic dimensions to 8x8 to ensure we have a reasonable size arrays per partition when scaling out. The color of each point denotes the normalized stall free run

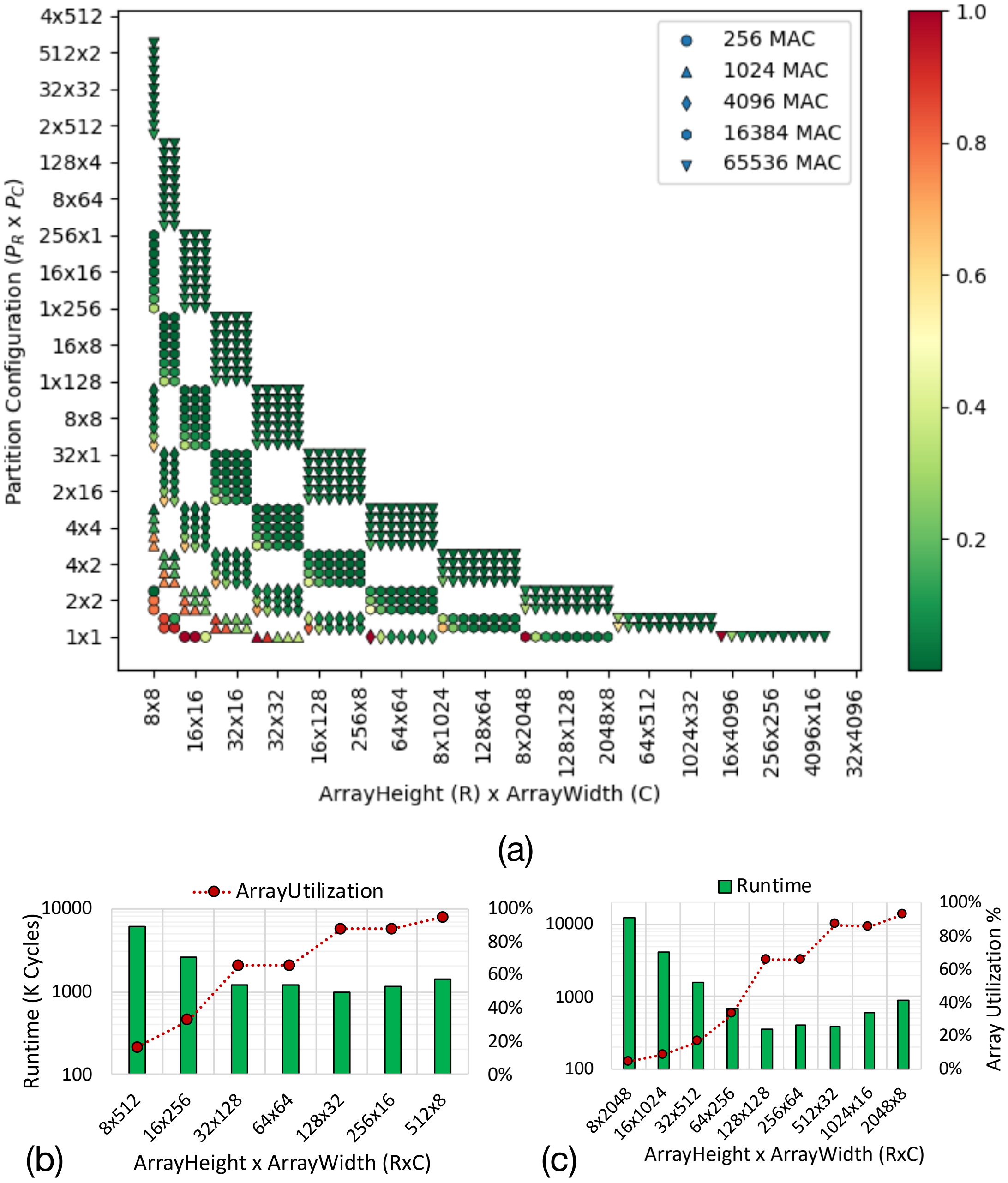


Figure 3.9: (a) The search space of all possible scale-up (monolithic) and scale-out (partitioned) configurations, with different array sizes; the color represents runtime for TF0 layer of the Transformer model, normalized to max runtime across configurations for a given array size. The variation in runtime and array utilization for all scaled-up configurations when running TF0 layer for (b) 2^{14} MACs, (c) 2^{16} MACs.

time when TF0 is run using OS dataflow. Run times are normalized to the highest runtime among all the configurations for a fixed number of MAC units.

Effect of Aspect Ratio on Scale-up Array. From this chart we can get a first order estimate of runtime variation between partitioned and monolithic configurations. We observe that the highers runtimes are usually located near the points corresponding to y value of 1×1, which represent the monolithic configurations. Figure 3.9(b-c) depicts the various aspect ratio (Row:Column) configurations for monolithic arrays with 4096 and 16384 MAC units respectively. The first observation is that, the difference in runtime for optimum configuration and others can vary by several orders of magnitude even when the workload is the same, depending on the size of the array. In fact, with larger arrays this difference is exacerbated. Second, the aspect ratio of the optimal configuration is not the same at different performance points, necessitating the need to have a framework to examine various configurations. When considering the array utilization, another interesting trend arises. For configurations with low array utilization, the runtime of the layer is high, which is expected. Also, runtime generally drops with array utilization. Interestingly, when the array dimensions become significantly rectangular, the effect of utilization is less pronounced. In these configurations even though a high utilization is achieved, the improvement in runtime is minimal. This is due to the fact that the time to fill in and take out the data starts dominating, as captured in Equation 3.3.

Comparison of Best Runtime. Moving to the points up along the y-axis in Figure 3.9(a) show almost monotonic improvement in performance, depicting that partitioning is always beneficial. To further investigate this trend in Figure 3.10 we plot the stall free runtimes corresponding to the fastest scaled out (monolithic) configuration normalized to the lowest runtime achieved among all the scaled-out (partitioned) configurations using equal MAC units. Figure 3.10(a) plots the rations for first and last five convolution and fully connected layers of Resnet50 CNN for different number of MAC units. It can be observed that monolithic configurations are sometimes significantly slower (25x for CB2a_1 layer)

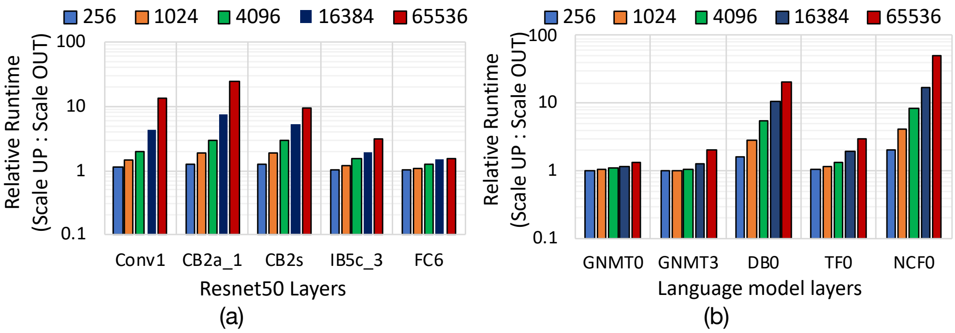


Figure 3.10: Ratio of no stall runtimes obtained in best scaled-up array configuration vs best scaled-out (partitioned) configuration for a few layers in (a) Resnet50 and (b) Language models, for different MAC units

that partitioned configurations, and never faster that the corresponding partitioned configuration. Moreover, for a given layer, the relative slowdown tends to amplify when the hardware is scaled. This trend is also replicated in language models, which predominantly use fully connected layers as seen in Figure 3.10(b). Here for 65536 MAC units the best monolithic configuration is 50x slower than the best partitioned configurations.

Note that since the runtimes involved in the above charts are stall free, the memory is not involved in slowdown. Therefore, the root cause of this slowdown can be understood by a closer look into the analytical model. First we should remember that in both monolithic and partitioned configurations the amount of serial computation is equal assuming all the MACs are utilized, or in other words the number of folds are equal. However from Equation 3.4 we can see that the runtime per fold is directly proportional to the array dimensions. Which explains the trend that the partitioned configurations are always faster. Furthermore, the difference in runtime per layer is amplified if the number of folds are high, even when both the arrays are fully utilized and the difference comes from data loading and unloading times. Also, utilizing the entire array in a monolithic configuration, howsoever flexible, is often not possible, as we can notice in Figure 3.9(b-c), which limits the amount of available compute resources and thus, contributes further to the relative slowdown.

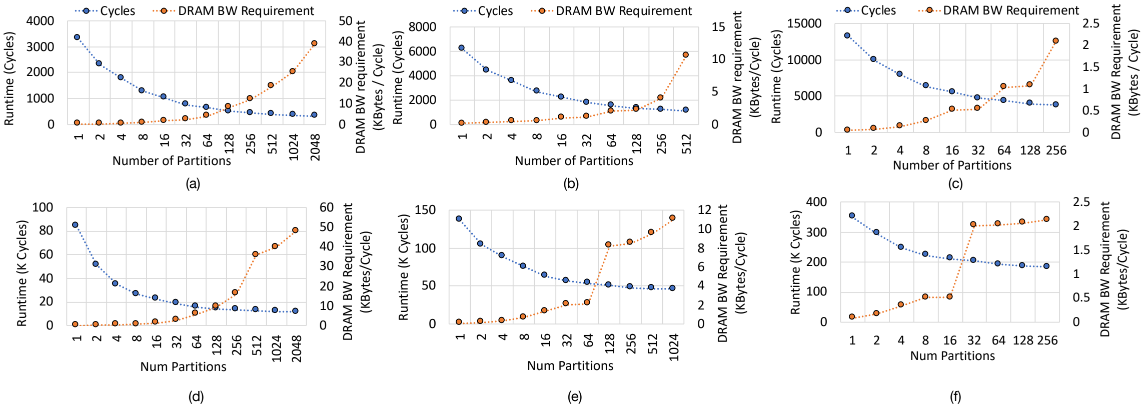


Figure 3.11: Trends for best possible stall free runtime and DRAM bandwidth requirements when the number of partitions are increased from monolithic array in CBa_3 layer in Resnet50 for (a) 2^{18} MAC units, (b) 2^{16} MAC units, and (c) 2^{14} MAC units; and TF0 layer in Transformer for (d) 2^{18} MAC units, (e) 2^{16} MAC units, and (f) 2^{14} MAC units

3.4.1 Cost of scaling out

Observations from the experiments in the previous section seem to suggest that scaling out is the best strategy to achieve the optimal runtime. However this choice involves paying additional costs as we discuss below.

The immediate cost of a partitioned design is the loss of spatial reuse. In a big systolic array any element read from the memory is used by processing elements along a row or column by forwarding it on the internal links of the array. Dividing up the array into smaller parts reduces the number of rows, or columns, or both, resulting in drastic reduction of this reuse opportunity. This is then reflected in terms of number of SRAM reads, data replication, and the input bandwidth (BW) demand from the DRAM. The loss of reuse within the array over short wires also leads to longer traversals over an on-chip/off-chip network (depending on the location of the partitions) to distribute data to the different partitions and collecting outputs - which in turn can affect overall energy.

Runtime vs. DRAM BW Requirement. In Figure 3.11 we plot the DRAM BW requirement and runtime for layer CBa_3 in Resnet-50 and layer TF0 in Transformer, as a function of number of partitions, for given number of MAC units. For all the three cases a total of 512KB of SRAM is allocated for IFMAP buffer, 512KB for Filter buffer, and 256

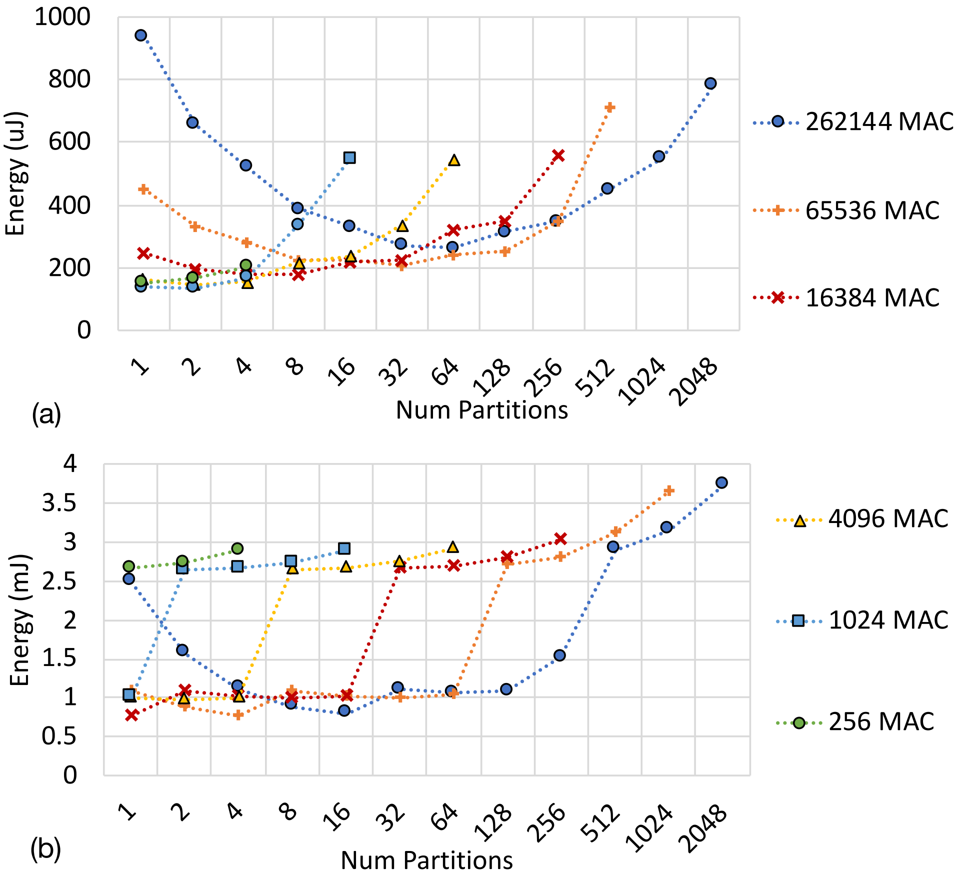


Figure 3.12: Energy consumption in running (a) Layer CBa_3 from Resnet50 and (b) layer TB0 from Transformer, when scaling-up and scaling out with different MAC units

KB for OFMAP buffer. This memory is evenly distributed among the partitions in case of scaling out. The BW numbers are obtained from our cycle accurate simulator when running the output stationary dataflow. As the number of partitions increase, the runtime goes down, however, BW requirements also rise due to loss of reuse originally provisioned by the internal wires, and increased replication of the data among the partitions, bringing down the effective memory capacity. The sweet spot lies at the intersection of runtime and bandwidth curves. When scaling to higher number of MAC units, it is interesting to note that the BW requirement is often higher than traditional DRAM BW. For instance, for both Resnet and Transformer layers with 2^{18} MAC units, about 10 KB/cycle of DRAM bandwidth is needed for stall free operation at the sweet spot.

Energy Consumption. In Figure 3.12 we study the effect of scaling out on energy.

Figure 3.12(a) depicts the energy consumption to run layer CBa_3 of Resnet50 as the number of partitions are increased for various MAC unit (barring the energy consumption of interconnection network). Figure 3.12(b) captures the same information for Layer TF0 for Transformer. For a given workload and hardware configuration, the energy consumption directly depends on the cycles MAC units have been active and the number of accesses to SRAM and DRAM. The counteracting effects of these factors can be observed in Figure 3.11, therefore lays down an interesting trade-off space. As the figure depicts, for lower number of MAC units (256, 1024 and 4096), the configuration with minimum energy is the monolithic configuration. However with increase in number of MAC units, the point of minimum energy moves towards the right of the chart, favouring more number of partitions. On other words the energy saved in by stealing runtime from powering the massive compute array is more significant than the extra energy spent by the loss of reuse. Furthermore, the bulkier the array, more the savings in compute to counteract the losses in reuse, which explains the observed trend.

To summarize the data indicates scaling out is beneficial for performance and with larger MAC units is more energy efficient that scaling up. However the cost paid is the extra bandwidth requirement to keep compute units fed, which even at sweet spots are significantly higher than the best scaled-up configuration for large MAC units.

3.4.2 Optimizing for multiple workloads

Any hardware accelerator should be performant for different workloads. To find such a globally optimized hardware accelerator, a global cost function must be minimized. However, as Figure 3.9(a) depicts even for a single workload as the global cost function is large and discontinuous. Optimally searching such a space for finding the global minima is out of the scope of this paper. Instead we propose a method to find reasonable pareto-optimal points for a given set of workloads

Considering the runtime as cost, our analytical model from Sec. Section 3.3.2 and

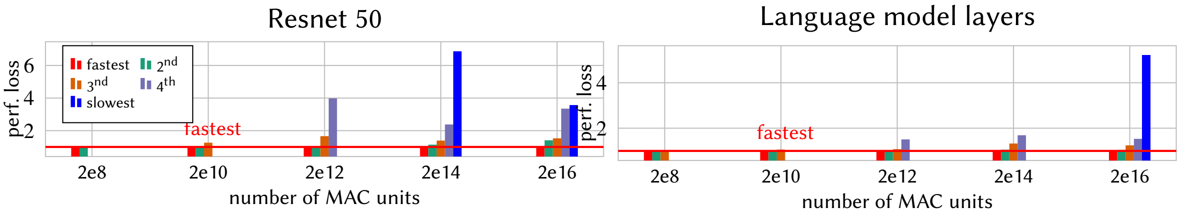


Figure 3.13: Total runtime loss vs. best configuration for *scale-up* ie. aspect ratio (R:C). Colors differentiate configurations ordered by runtime.

Section 3.3.3 or SCALE-Sim yields a runtime-optimal configuration, $a_k = (S'_C, S'_R, R, C)$, for each individual layer (i.e. workload $w_l = (S_C, S_R, T)$). We then search among these candidates for the globally optimized one, A. In case of runtime, the total runtime is additive and thus it is calculated by summing the runtimes T_r of all workloads w_l for each candidate a_k:

$$A = argmin_{a_k} \sum_{w_l} T_r(w_l, a_k)$$

As the number of candidates is limited, exhaustive search is feasible to find the optima.

In Figure 3.13 we plot the costs (runtime) of the various candidate configurations normalized to the cost of the pareto-optimal configuration obtained by the method mentioned above, for layers in Resnet50 and the language models mentioned in Table 3.4. In Figure 3.14 the normalized costs for all locally optimal candidates for scale-out is depicted. In both these cases we observe that the pareto optimal configuration is up to 8x faster than the locally optimal configurations. However, the second and third best configurations are within 20% for smaller number of MAC in both scaled-up and scaled-out configurations. However as the MACs increase the spread of runtimes and we see about 50% increase in runtime for second and third best configurations, while slower configuration taking several factors more cycles to complete than the best configuration.

3.5 Chapter Summary

This chapter discusses the simulation and analytical tools for systematically characterizing the systolic array based architecture design space. We first cover SCALE-Sim a cycle

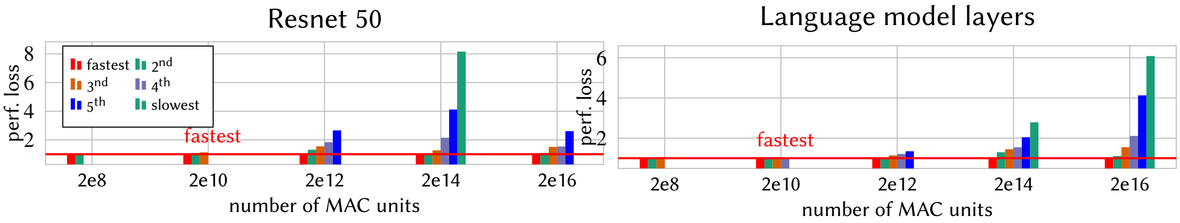


Figure 3.14: Total runtime loss vs. best configuration for *scale-out*. ie part order (P_R, P_C) and aspect ratio (R:C). Colors differentiate configurations ordered by runtime.

accurate systolic array based DNN accelerator simulator, and describe the inputs and outputs that are used and generated by the tool. This chapter also provides details on the scope of the tool in terms of designs, mappings, and the workloads that the tool can work with. Next we develop and describe an analytical model for systolic array based accelerators, which helps expose the various architectural and workload parameters that effect the performance and efficiency when GEMM workloads are executed on the systolic array based accelerator.

The utility of the analytical model is that, it allows designers to statically identify the key architectural parameters that contribute the most for performance and energy efficiency for a given workload before any simulations are run. This helps prune the search space for the simulation runs to find the optimal parameter values. The SCALE-Sim simulator on the other hand, provides plethora of fine grained execution information without explicit functional simulation, saving significant time per simulation. The cycle accurate traces generated by the tool also helps in fine tuning the design of system level integration. Finally, in this chapter we demonstrate the utility and convenience of the tools by performing a study to find the optimal accelerator design at scale.

CHAPTER 4
INTERCONNECT AWARE SCALABLE DNN ACCELERATOR IMPLEMENTATION USING HARDWARE CASCADES IN XILINX FPGAS

Note: The work and materials presented in this chapter has been developed in collaboration with Dr. Nachiket Kapre and his group at University of Waterloo.

4.1 Introduction

In the previous chapter (Chapter 3), the discussion was focused on building scalable accelerators from an analytical perspective. One important observation presented in the chapter is that various array configurations are optimal for various layers of the workload (Section 3.4). Traditionally ASIC based accelerator design is optimized for the common case with some or no reconfigurability available to accommodate for different workloads. An alternative approach to counter this limitation is to implement the accelerator design on an FPGA such that dynamic reconfiguration is attainable by either exploiting the dynamic reconfigurability of the interconnect or the logic fabric itself. The penalty of this approach however is the compromise in clock frequency which is a natural consequence of emulated logic.

In this chapter, I describe a design for scalable accelerator implementation on Xilinx VU37P FPGA, which is the work performed by me and my collaborators. This work circumvents the limitations of emulated fabric of FPGAs by constructing a configurable accelerator using hard DSP48 and memory cascades already present in Xilinx Ultrascale+ FPGAs. Figure 4.1 depict the schematics of UltraRAM (URAM), BlockRAM (BRAM), and the DSP48 block interconnected in hardware cascades. In this work we demonstrate the construction of compute units for computation and efficient data reuse for convolution and matrix-vector operation using the cascaded DSP and memory units. The entire accelerator

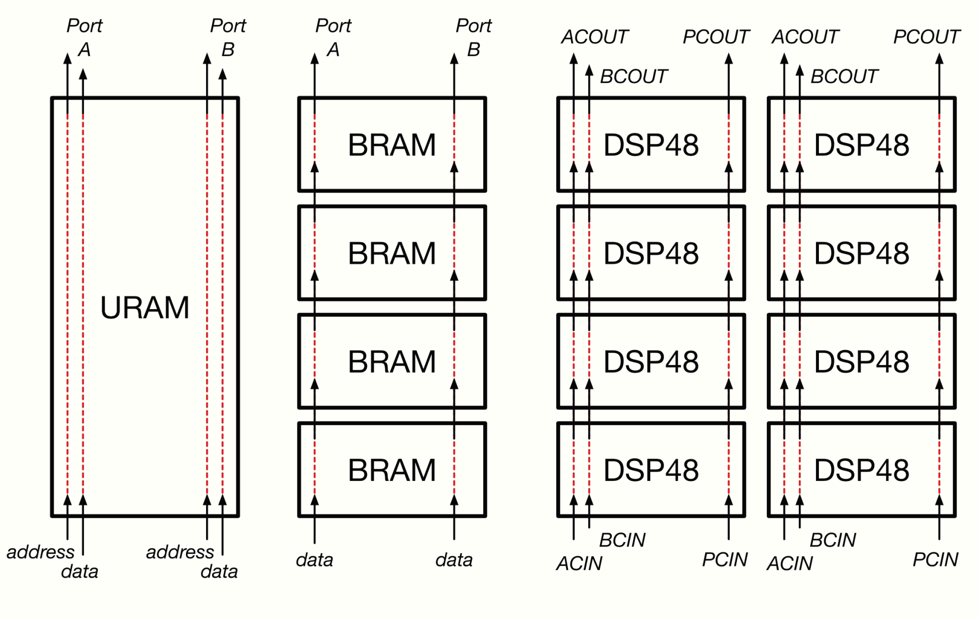
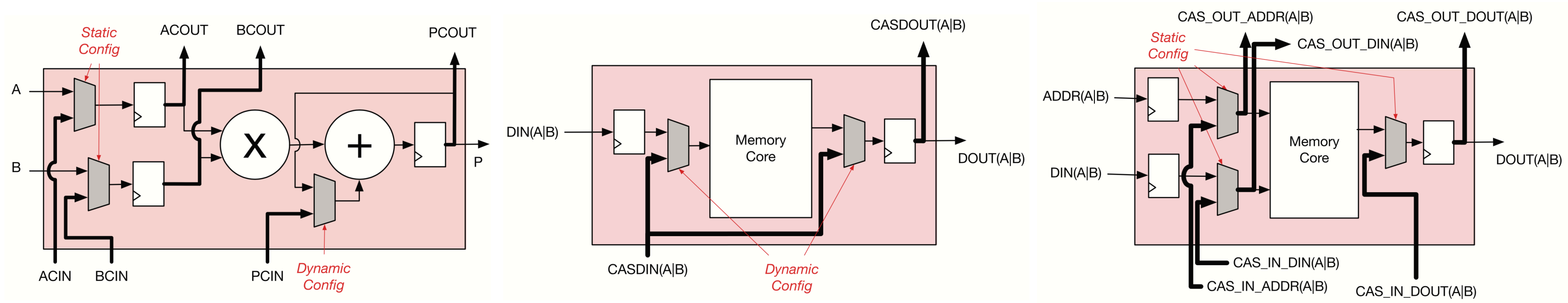


Figure 4.1: High level view of cascades connections between DSP48, RAMB18, and URAM288 blocks

is then constructed by augmenting these units with each other as described in Section 4.3. The flexibility of an FPGA allows us to determine the ratio of area to be allocated to convolution and matrix-vector multiplication unit during deployment for a given workload, thus enabling tailored acceleration. On the other hand, using only the hard logic and interconnect structures on the FPGA allows us to achieve high frequency, close to the max frequency of the board as depicted in Table 4.2.



(a) DSP48 Cascade (891MHz)

(b) RAMB 18 Cascade (825MHz)

(c) URAM288 Cascade (650MHz)

Figure 4.2: Hard cascades structures embedded in Xilinx hard blocks

4.2 Background

4.2.1 Dedicated Cascade Interconnect in Xilinx Ultrascale FPGAs

The Xilinx UltraScale + device family , integrates thousands of hard resources such as DSP and RAM blocks in a columnar arrangement. We enumerate the salient features of the different blocks below:

1. **DSP48**(Figure 4.2(a)): These components primarily support arithmetic integer operations including 27x18 multiplication, and 48b accumulation. A unique feature of the Xilinx DSPs is that they expose configurability within the DSP block to the FPGA logic fabric for runtime control. A developer may not only choose the kind of operation being performed in the DSP block, but also change data routing and data movement pathways within the DSP. The key feature of the DSP48 blocks we wish to exploit in this paper is the ability of multiple DSP blocks to cascade together in a chain-like configuration. This is supported either for performing accumulation of a series of partial products (adder cascade), or for permitting efficient data reuse for certain inputs (input cascade). While the adder cascade is programmable dynamically, the input cascade is only statically configured.

2. **RAMB18** (Figure 4.2(b)) : Modem FPGAs provide access to thousands of small distributed on-chip memories that have configurable port widths, and other statically configurable operating modes. A particularly unique feature of the Xilinx UltraScale FPGAs is the presence of nearest-neighbor, dynamically cascade-able connections for the two data ports in the same direction as the DSP cascades (uphill). This allows the developer to construct deeper memory structures, cascaded FIFOs, and other user-configurable dataflow patterns. The multiplexers controlling dataflow are only available on the data ports.

3. **URAM288** (Figure 4.2(c)) : UltraScale + FPGAs introduced higher density SRAM blocks with 288 KB capacity that sacrifice port width flexibility for lower cost. While the port aspect ratios are not programmable, there is still a column-spanning cascade network for the data ports along with address. This allows the developer to address any location in any URAM block in a column with ease. In contrast to BRAM cascades, the URAM cascade network separates the read and write ports into independent cascades. Importantly, it provides cascade-ability for data, address, as well as control signals.

4.3 High-Frequency FPGA Cascades

The architecture of FPGA-based ML accelerator in this paper exploits (1) the resource balance constraints of the device, and (2) unique cascade interconnect features of the UltraScale + family. In this section, we first discuss the building blocks for Convolution and Matrix-Vector multiplication blocks and show how to map these over the cascades to create repeating tiles that balance capacity, bandwidth, and precision. After that, we provide an overview of the design space of possible implementations. Pooling operations are mapped onto the same resources as the Convolution engines, while ReLU softmax operations are provided as by passable operators prior to data commit.

4.3.1 Building Block: Convolution

We present a weight-stationary implementation of convolution that takes advantage of data reuse patterns in convolution without transforming it to memory-hungry matrix-matrix operations as discussed earlier.

For our accelerator template, shown in Figure 4.3 , we take a 3×3 convolution and parallelize the inner convolution loops across a series of nine DSP48 blocks, four BRAMs, and two URAMs. The nine multiplications and eight additions are mapped to the nine DSPs in a sequential chain fashion. Each DSP48 computes the result of multiplying a weight

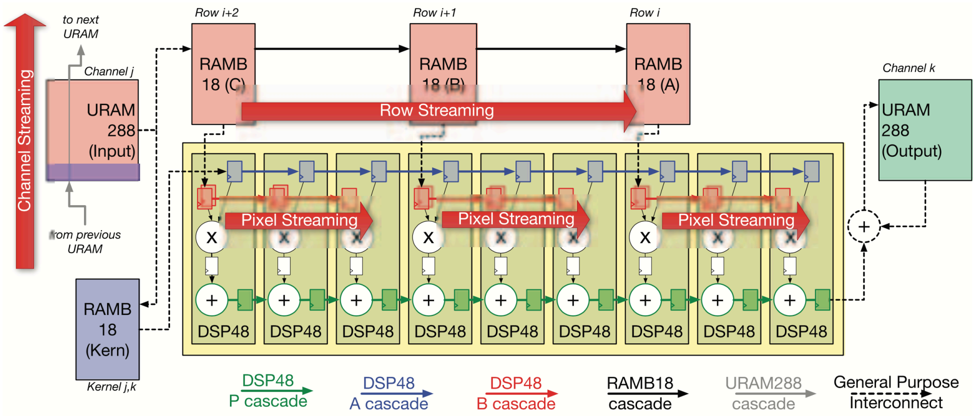


Figure 4.3: Design of a 3 × 3 convolution block. DSPs configured in SIMD=2 mode, a set of 8bit weights are shifted into the B cascade. One stream of $2 \times 8b = 16b$ data streamed in B cascades from different rows. BRAM cascades also configured to exploit row reuse

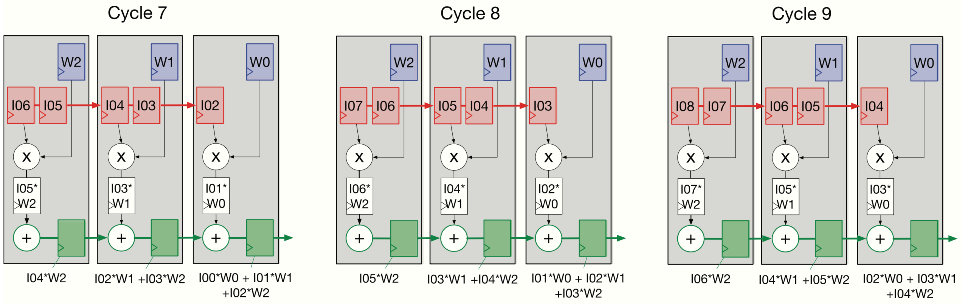


Figure 4.4: The state of the DSP48 blocks in steady operation for 1 × 3 filter slice using 3 DSP48 units.

with a corresponding input value and computes the partial sum of products. We operate the DSP in a $SIMD = 2$ mode, i.e., we configure the 16b multiply and add datapath into two parallel 8b operations to enhance throughput. The DSP 48P cascade (Figure 4.2(a)) is used to accumulate the result of nine multiplication results.

We show snapshots of the internal DSP48 state in Figure 4.4 after cycle 7, 8 and 9. In the first six cycles, we initialize the pipelines from their empty state. We pipeline output of each multiply and add operation and orchestrate the input pixel shifting using a 2-stage pipeline to align data for systolic operation. The nine weights of a kernel are loaded into the B cascade chain of the DSP48 block and locked in place. The kernel BRAM store multiple sets of kernel weights that are accessed infrequently. The inputs are streamed over the A

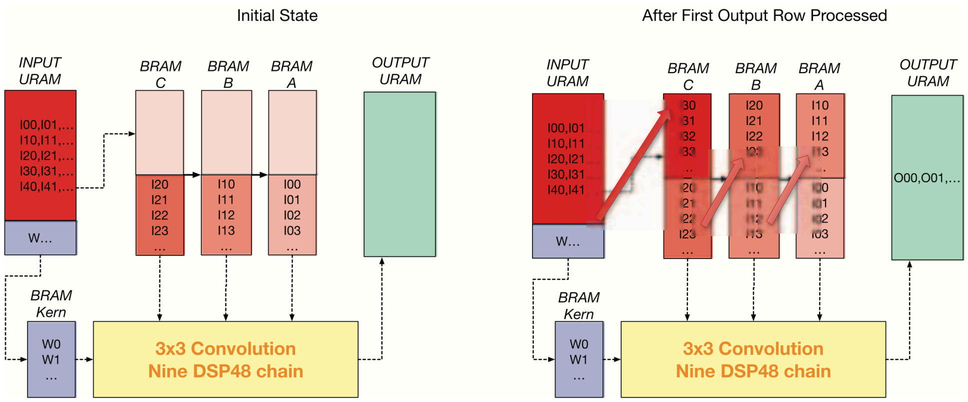


Figure 4.5: Data movement between URAM and BRAM to support 3×3 convolution while exploiting data reuse

chain of the DSP48 blocks and split into three segments of length three. Using systolic pipeline mode of the DSP48 input chains, we are able to stream in a row and after an initial latency generate a stream of output pixels.

Each segment is fed by a single BRAM thereby requiring three BRAMs to stream row pixels into the DSP48 chain. We achieve input row data reuse by copying the row data into the next BRAM as its been read out. Thus, in double- buffered fashion, the row data stages its way through the three BRAMs feeding into the three A segments. This is illustrated via snapshots of the row BRAMs in Figure 4.5 . Three rows are fed in parallel to the Convolution Block while simultaneously being shifted into the next BRAM via the hard cascades. New rows are streamed from the URAM and one row is updated in the output URAM.

4.3.2 Building Block: Matrix-Vector Multiply

Unlike convolution, matrix-vector computations have low arithmetic intensity and require blocking to support diverging problem sizes across layers of the network.

Our accelerator design, shown in Figure 4.6 , parallelizes the computation across a series of nine DSP48 blocks. The multiply-accumulate P cascade chain is identical to the one used in the convolution block with the exception of SIMD =1 mode configuration.

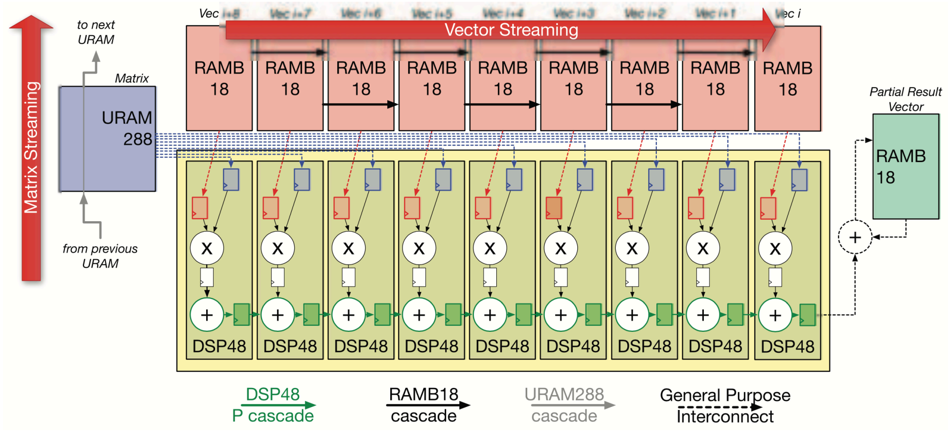
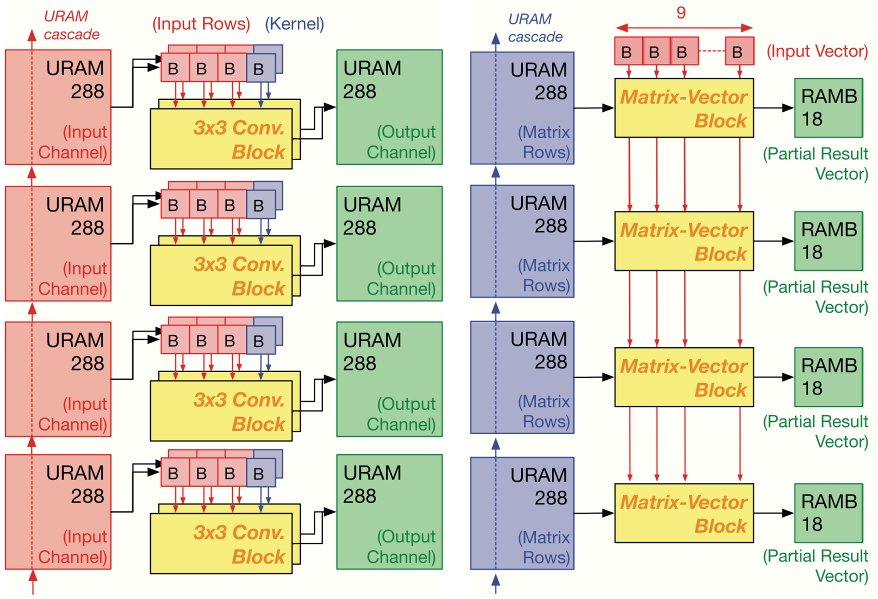


Figure 4.6: Design of a Length-9 dot product unit that can perform URAM-capacity-limited matrix-vector multiplications. The DSP-48 chain is configured with SIMD=1 mode to perform matrix-vector product of 8b inputs. A chain of 9 DSPS is configured to perform length-9 dot products. URAM distributes 9 chunks of 8b values from the matrix in a row-wise fashion. The bank of 9 BRAMs distributes 8b values.

Nine BRAMs feed 8b data to the adjacent DSP48 blocks to provide the vector input. The URAMs support 72b port widths, which are sliced across the nine DSP48s to distribute the matrix entries. The final result is accumulated in the result BRAM. The result is distributed across the FPGA via BRAM cascades so the output vector can be fanned out in preparation for the next computation.

4.3.3 Scaling And Tiling

The final FPGA organization that uses these building blocks for tiling must consider the unique DSP-BRAM-URAM capacity and bandwidth balance available on the UltraScale + device family. We must also consider the deep network architecture as layer connectivity must be considered to ensure data movement between layers in handled properly. The Xilinx VU37P FPGA enforces a resource balance of 1 URAM288, 4.2 BRAM18s, and 9.4 DSP48s. Additionally the URAM can supply 72b data, while the RAMB18s can supply 18b data in True Dual Port (TDP) mode. The DSP48 blocks can consume inputs to feed the 27x18b multiplier and can be configured in SIMD =2 mode to compute two 8x8 multiplications with a common input.



(a) 3×3 Convolution Tiles (b) Matrix-Vector Multiply Tiles

Figure 4.7: Repeating tiles of the ML accelerator that obey Xilinx VU37P resource, capacity bandwidth constraints: Two Convolution tiles sharing the weight memory, while 4 tiles of Matrix-Vector multiplication block share the vector RAM

We design repeating tiles to satisfy the resource-balance, bandwidth, and precision constraints of the FPGA device:

1. **The Convolution Tile:** As shown in Figure 4.7(a) , each URAM supplies input channel data to two convolution blocks instead of one. The URAM has enough bandwidth to satisfy the needs of both blocks and helps create a repeating tile with 2 URAMs, 8 BRAMs, and 18 DSPs which is well within the VU37P balance of 2 URAMs, 8.4 BRAMs, and 18.8 DSPs. In addition, the URAM read/write cascades are employed to move the input channels across the different convolution tiles. This is necessary to support the all-to- all communication pattern inherent in an implementation of a convolution layer - here, each input channel convolves with a unique kernel for each output channel combination. We stream the input channel into a convolution block to update a particular output channel which is resident in the output URAM of that block. Simultaneously, the input channel is the shifted into the next URAM for the next output channel computation using the dedicated URAM cascade wiring.

This all-to-all pattern is thus implemented by shifting data along a ring configured out of the URAM cascade structures.

2. **Matrix-Vector Multiplication Tile:** In Figure 4.7(b) , we see that, we again need to generate a resource, bandwidth, and precision aware repeating tile. In this scenario, each URAM with its 72b port distributes matrix data to nine DSP48s in 8b chunks. We configure nine BRAMs to supply 8b vector data in parallel to the nine DSP48 blocks. Since the vector is common across all dot product evaluations, we fanout each BRAM output to the different copies of the Matrix- Vector multiplication blocks. Thus, each repeating tile has 4 URAMs, 9 BRAMs (input)+4 BRAMs (output), 36 DSP48s which is well within the VU37P resource balance of 4 URAMs, 16.8 BRAMs, and 37.6 DSPs. If multiple FC layers are to be sequenced together, the resulting partial vector outputs stored in the output BRAMs are then shifted in a ring-like fashion across the multiple tiles to replicate the output vector across all tiles. This will the allow the next FC layer computation to proceed in an identical fashion.

4.3.4 System Design Strategy

Finally, we determine the use of Space-Division multiplexing as the high-level parallelization strategy for supporting the different layer configurations on the same FPGA. We can partition the FPGA statically into two regions: one for convolutions, and another for matrix-vector multiplication. We can calibrate the balance based on the specific requirements of the deep network architecture (GoogLeNet splits across 80% convolution, and 20% matrix multiplication). To limit resource idling at the cost of inference latency, we (1) replicate the design to evaluate multiple images in parallel, or (2) decompose the FPGA into sub-regions devoted to a subset of layers of a CNN. Unlike the Xilinx SuperTile , overlay, our design generalizes to a range of benchmarks beyond GoogLeNet. Reconfiguration was ruled out due to an exorbitant 140 ms of programming time [51]

Table 4.1: MLPerf and GoogLeNet benchmark characteristics.

Topology	**Operation Count**			**Storage** (bytes)	
(MLPerf)	All	Conv	MM	∑Wts.	Activ.
AlphaGoZero	352M	352M	353K	1.5M	92K
DeepSpeech2	1.7G	1.7G	74K	355K	6.5M
FasterRCNN	3.5G	1.6G	1.8G	13M	802K
NCF	11M	0	11M	11M	138K
Resnet50	3.4G	1.6G	1.8G	25M	802K
Sentimental	210M	0	210M	172K	30.7M
Transformer	113M	35M	78M	77M	4096
GoogLeNet	1.3G	1.3G	46M	6.8M	200K

4.3.5 Overall FPGA Architecture

The Xilinx VU37P FPGA supports HBM interfaces with 32× AXI ports with 256b 450 MHz rates feeding into the FPGA core. This is used for initial loading of the on- chip memory contents and is not needed thereafter for all benchmarks (except Transformer, see Table 4.1 later) as the weights and worst-case activation state is $<$35 MB. It also includes 960 URAM288 blocks, 9024 DSP48 slices, and 4032 RAMB18k blocks. Our architecture can support 960 computing blocks configured as 480 3×3 Convolution tiles, or 240 Length-9 Matrix-Vector Multiplication tiles. The Convolution tiles perform two 3×3 convolutions across 18 DSP48 blocks configured in $SIMD = 2$ mode thereby processing 36 $8b \times 8b$ multiplications and 36 24b accumulations per cycle. The Matrix-Vector Multiplication tile can process four dot products of length 9 to yield a throughput of 36 $8b \times 8b$ multiplications and 36 48b accumulations per cycle.

4.4 Methodology

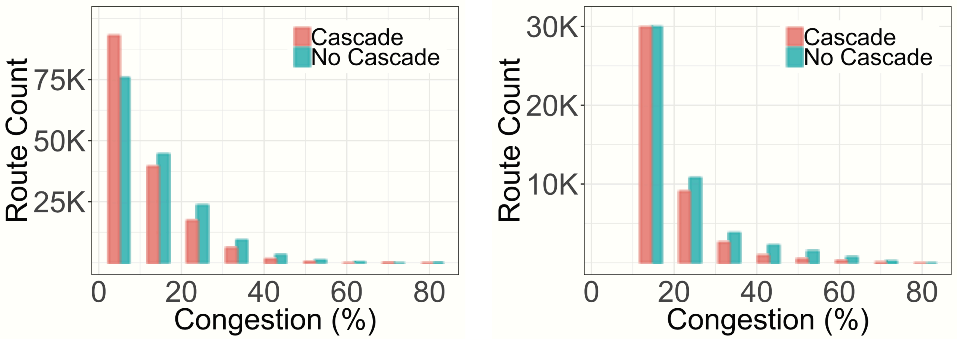
4.4.1 FPGA Mapping

We describe our designs directly in RTL component-level instantiations of DSP48, RAMB18, and URAM288 blocks and associated controllers for orchestrating data movement and control flow for convolutions and matrix-vector multiplication. We use Vivado 2018.2 for our experiments and use a tight 1 ns timing constraint for the CAD tools. We generate explicit physical location mapping for the DSP, and RAM components, and supply customizable pipelining to enforce the high-frequency design constraint. We measure the resulting frequency of the mapped design, and interconnect utilization metrics to quantify the extent of wiring reduction. We map our designs to the Xilinx UltraScale + VU37P FPGA XCVU37p-3.

4.4.2 Performance Analysis Of MLPerf Benchmarks

We build a cycle-accurate model of our design using an open-source CNN accelerator simulator from ARM called SCALE-Sim . We model our system as a 9×1920 systolic array for Convolution and 9×960 array for Matrix-Vector Multiplication. Each grouping of 9 DSP48s in each chain are modeled as ID systolic chains. We are able to model a variety of dataflows including the "weight-stationary" and "input-stationary" models for our design. For convolutions, our modeling framework includes support for parallelization of partial sum generation for an output channel. This feature can be added to our RTL design with a minor modification requiring an adder chain across multiple convolution blocks.

We run DNNs from MLPerf to evaluate the performance of our cascade design along with GoogLeNet. We validate the cycle counts for various layers in GoogLeNet against that reported by our RTL simulation and SCALE-Sim runs. In Table 4.1 , we tabulate the peak memory usage footprint of MLPerf workloads that includes sum of all weights (filters and matrices) as well the worst-case activation layer storage costs. With the exception of Transformer benchmark, we never exceed the 35 MB capacity of the 960 URAMs on the VU37P.



(a) Convolution Congestion

(b) Matrix-Vector Congestion

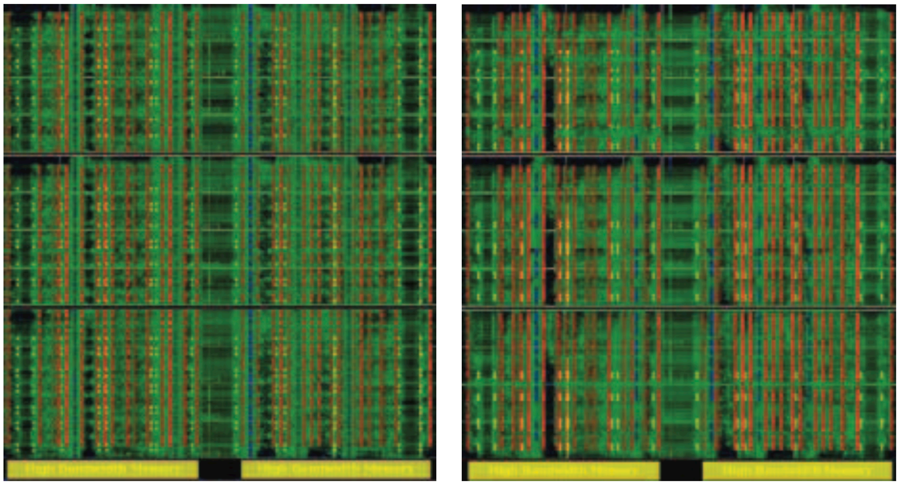
Figure 4.8: Histogram of congestion of routes for full-chip Convolution and Matrix-Vector multiplication hardware

4.5 Evaluation

We now discuss the implementation results of our interconnect-aware mapping of ML problems on Xilinx UltraScale + VU37P FPGA. We first highlight the frequency and utilization of our proposed cascade design against one where the data movement is directly mapped over the soft fabric instead along with related work. We then discuss performance results for the MLPerf benchmark set for our platform and use GoogLeNet benchmark for comparison against Xilinx SuperTile.

Table 4.2: Resource and Frequency Trends for Convolution and Matrix-Multiplication blocks, tiles, and full-chip layouts

Design	Size	LUTs			FFs			Clk (ns)			Net Util. %		
		Fabric	Cascade	%	Fabric	Cascade	%	Fabric	Cascade	%	Fabric	Cascade	%
Convolution	Block	325	327	0%	1.3K	1K	30%	0.9	0.9	0%	0.01	0.01	0%
	Tile (2 blocks)	424	435	-2%	1.9K	1.5K	26%	0.9	1	-10%	0.02	0.02	0%
	Full Chip	20.9K	21.1K	-1%	95.3K	71.2K	32%	1.4	1.4	0%	12.8	9.6	33%
Matrix-Vector Multiplication	Block	98	98	0%	775	688	12%	1	0.9	10%	0.01	0.01	0%
	Tile (4 blocks)	375	374	0%	2.3K	1.9K	21%	1.1	0.9	22%	0.04	0.05	-8%
	Full Chip	90.2K	90.2K	0%	56.8K	46.6K	21%	1.5	1.3	15%	9.3	8	16%



(a) Convolution Layout (b) Matrix-Vector Layout

Figure 4.9: Full chip VU37P layout of Convolution and Matrix-Vector multiplication hardware

4.5.1 Frequency Trends

In Table 4.2, we show the LUT and FF cost of the various design configurations along with frequency and interconnect utilization data. As expected, our careful bottom-up design methodology delivers high performance outcomes with the worst clock period of 1.5 ns. When considering the use of cascaded interconnect structures we observe a 15% reduction in clock period and 20–30% reduction in FF use. Convolution designs do not show much clock frequency improvements as our designs are extensively pipelined even without cascading features. A key measurement is the interconnect utilization drop of 16–33%. This is directly attributable to the use of cascade rather than general purpose interconnect for data movement.

We show the effect of cascading on network congestion through the histogram plots in Figure 4.8 . It is clear that the cascaded design uses fewer congested routes than the non-cascaded design. This gap is stark for Convolution design with as many as 20% more routes in the lowest congestion bin. At the far end of the spectrum in the highly congested bins, we have 5–10× fewer routes for the cascaded design configuration.

We show chip-spanning FPGA layouts of our Convolution and Matrix-Vector Multiplication designs in Figure 4.9 . The regularity of the connectivity, and the use of nearest-neighbour cascade resources are visible in the layout. Complete FPGA mapping takes 6–7 hours on Vivado 2018.2 for these designs and is able to get close to the timing target of a ns. This easily outperforms the 250 MHz operating frequency of Brainwave design. Our implementation frequency is limited purely by the hard resource constraints than our design architecture. We ran an experiment by removing the URAM operations from our netlist and found the peak frequency achievable is 800900 MHz in agreement with the limits of the DSP and BRAM components.

When compared to related work, our 650 MHz clock is within 70 MHz of the state-of-the-art 720 MHz Xilinx SuperTile [52, 53] design, and much faster than other contemporary designs. While Xilinx SuperTile only uses 56% of the DSP48 blocks, we use 95% of our DSP48 resources delivering an effective throughput that is $\frac{100\%}{56\%} \times \frac{650MHz}{720MHz} = 1.6\times$ better. Brainwave operates between 225–500 MHz on the Stratix V-Stratix 10 silicon and is constrained by the memory controller interfacing speeds. In contrast, we operate like SuperTile by keeping weights and activations fully on-chip in URAM288s and loading only once at the start.

4.5.2 Performance Trends

First, we tabulate the inference latency and throughput results in Table 4.3. We see runtimes $<$ 2ms across all benchmarks in the MLPerf set. In particular, we highlight the runtime of GoogLeNet at 0.4 ms which outperforms the 3.3 ms latency of the Xilinx SuperTile design on the VCU1525 board with the VU9P-2 FPGA card (30% fewer DSP48s and 10% more BRAMs and identical URAM counts compared to VU37P FPGA). Our design uses almost all the DPS48s rather than the 56% of the SuperTile design, and also operates everything at the 650 MHz identical clock rather than half rate RAM speeds of SuperTile. When considering throughput, the SuperTile array offers an impressive 3K images/sec of processing capacity. Our design can deliver a peak throughput of 2.4K images/sec which is 25% lower than

Table 4.3: Xilinx VU37P FPGA inference latency (ms) and throughput (inf/s) for MLPerf benchmarks and GoogLeNet.

Topology (MLPerf)	**Ratio** (Conv:MM)	**Cycles**	**Time** (ms)	**Tput.** (inference/s)
AlphaGoZero	90:10	60K	0.09	10K
DeepSpeech2	60:40	1.2M	1.89	528
FasterRCNN	30:70	903K	1.38	719
NCF	0:100	2.4K	0.003	260K
Resnet50	30:70	848K	1.3	766
Sentimental	100:0	24K	0.037	27K
GoogLeNet (Us)	70:30	261K	0.40	2.4K
GoogLeNet (SuperTile [53])	-	-	3.3	3K

SuperTile. The Xilinx design maximizes device utilization by pipelining execution across layers with three identical copies of the design, each with a chain of four processors sized differently and working on subset of the DNN layers. This sacrifices latency but improves throughput by allowing each processor to maximize device utilization. In contrast to the 95% utilization achieved by SuperTile, we only achieve 50% utilization but deliver superior inference latency.

In Figure 4.10 , we show the benefits of systematic resource allocation of the FPGA to the different layers of the neural network. We divide resources to convolution:matrix-vector portions (the x-axis ratio shown in Figure 4.10) of the application keeping overall FPGA design area at 100%. Our current goal is to optimize for inference latency, and the particular balance of resources sacrifices some throughput to deliver superior inference latency outcomes. The resource balance at the runtime minimum point matches the ratio of work performed in the Convolution and Matrix-Vector Multiplication phases as indicated in Table 4.1 . Resnet50 and FasterRCNN workloads shows a trade-off that suggests best performance in the 30:70 resource division ratio. Rest of the workloads end up preferring a solution that is on either ends of the resource balance scale.

Finally, in Figure 4.11, we show a strong correlation between the problem requirements

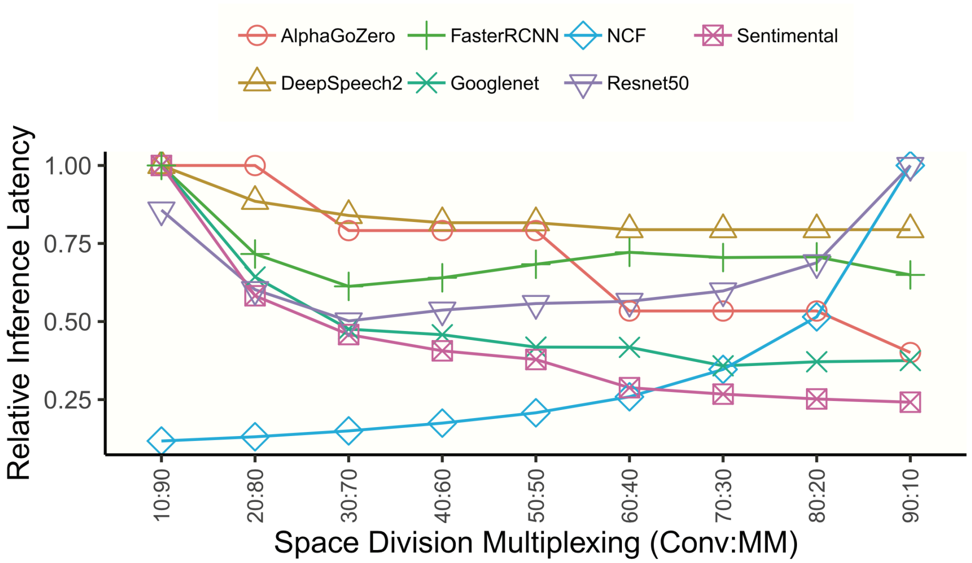


Figure 4.10: Optimizing resource allocation for MLPerf Workloads

of the MLPerf benchmarks and the division of hardware resources to Convolution or Matrix- Vector multiplication tiles. Our optimization shows that there is a clean transition between the use of Convolution to Matrix- Vector multiplication hardware allowing us to stream the images through the chip.

4.6 Lessons

Based on our study, we identify the following suggestions for future ML-friendly FPGA designs:

1. **URAM Bandwidth Balance:** A 2× improvement in URAM memory bandwidth from each URAM will help address the memory bandwidth bottleneck for the matrix-vector multiplication phase of the computation.

2. **Dynamic Programmability:** For the Xilinx DSP48s, the cascades on the AB inputs, fracturing modes, and URAM cascades remain stubbornly statically configurable. This forces a designed to lock down data movement patterns at compile time and tailored uniquely for either convolution or matrix-vector multiply phases which

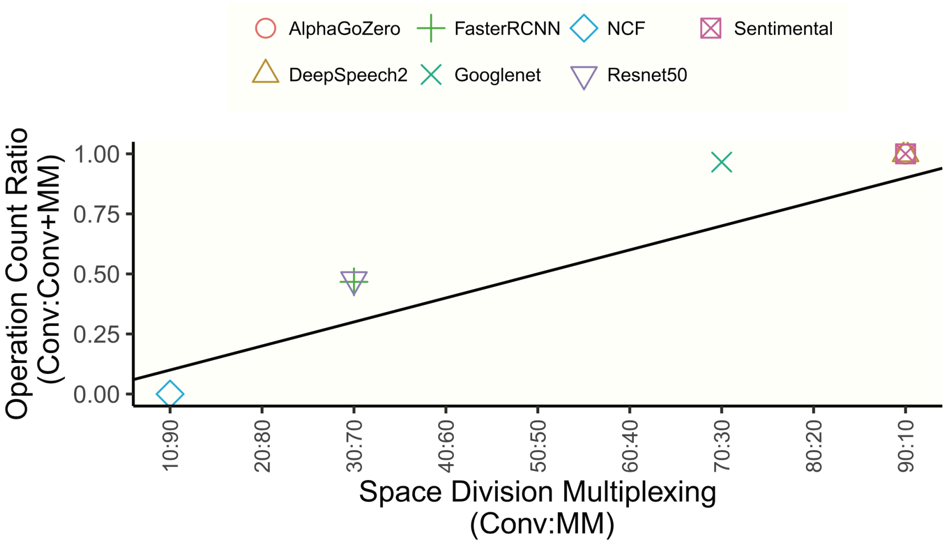


Figure 4.11: Correlating MLPerf benchmark characteristics to Space Division Multiplexing arrangement

makes a unified full chip design is difficult.

3. **Impact of Xilinx Versal FPGA:** As stated earlier, we propose a closer look at existing hard interconnect structures within the FPGA fabric rather than embracing rigid ASIC- like computing elements targeting only the AI application domain. This departure also imposes a high design cost on the developer through the adoption of a mixed RTL and C/ C++ or VLIW-assembly programming. The Versal system-level hard NoC does not address the intra-accelerator data movement requirements of computing workloads.

4.7 Chapter Summary

This chapter provides the details of flexible yet scalable DNN accelerator implemented in Xilinx VU-37P FPGA. The flexibility of the design is attributed to the fact that the design is implemented on a FPGA thus resource allocation can be determined in a case to case basis for different workloads. The scalability is the direct consequence of constructing the accelerator from the Convolution and Matrix-Vector multiplication blocks implemented using the DSP48 blocks, URAM and BRAM slices and hence connected by clever exploitation of existing hardware cascades. The novelty of this design lies in the fact that unlike traditional FPGA implementations, the accelerator is constructed using pre-exiting hard DSP48, URAM and BRAM cascades present in Xilinx Virtex FPGA. As depicted in Section 4.5, this lets us achieve high frequency operation, close to the maximum frequency leading to maximum performance while enabling flexibility.

CHAPTER 5
AIRCHITECT: LEARNING DESIGN AND MAPPING SPACE FOR CUSTOM ARCHITECTURES

5.1 Introduction

In the previous chapters, we have discussed about systematically capturing the design space using analytical models or simulators. The optimal points in this space are then determined by employing search algorithms which iteratively calculate the cost of the various points within the space. The optima, naturally is dependent on the target workload or the set of workload, and the implementation constraints.

Previous works have shown that the accelerator and mapping space for DNN workloads is extensive[26, 34, 35, 27] and understandably is non tractable when using brute force techniques. In face design space exploration (DSE) is one of the large contributors of the non recurring engineering cost (NRE) incurred when newer designs are needed to be implemented. Given the importance of the problem, there have been several recent proposals which try to make the search process fast and efficient using techniques like learned cost models [36], reinforcement learning [33, 35], genetic algorithms [34] etc.

Finding the optima using iterative cost estimation and search is a data driven decision making process. In this work, our hypothesis is that, it should be possible to automate this process, by learning the design space from the data obtained from simulation or analytical model queries by using machine learning methods. As we show later in this chapter, a model can be constructed and trained on the existing database of optimal design points, which when queried with workload parameters and design constraints is capable of predicting the optima in constant time.

In the subsequent sections of this chapter, we demonstrate that learning the architecture design space is feasible by using the three case studies dealing with finding the optimal architectural parameters and mapping strategies for a systolic array based DNN accelerator(see Section 5.3). The next section (Section 5.2) describes the case studies and the associated search space. We then perform a design aware and statistical analysis of the design space to understand the feasibility of the capturing the space using a learning model. Next, we describe our findings of formulating the design space exploration problem as a machine learning problem for different design tasks using the three distinct case studies(see Section 5.4). Finally we describe the custom designed neural network based recommendation model, which we found to faithfully capture the various design spaces pertaining to our case studies. We also present a comparative study of the performance of the model with the contemporary off the shelf machine learning models.

5.2 Case Studies

In order to evaluate our hypothesis on weather learning the design and mapping space of custom architecture is feasible, we chose three design tasks for systolic array based accelerators for GEMM workloads. The motivation to chose systolic array based design is simplicity and efficacy. The simple construction of the systolic arrays make it scalable, and easy to implement. Furthermore, systolic arrays are arguably the most efficient designs for accelerating GEMM workloads, which is perhaps why they are used in numerous DNN academic and industry DNN accelerator proposals. Owing to their simplicity and popularity, there is an abundance of literature studying these structures, including our work mentioned in the previous chapters. Using systolic arrays makes it convenient for us to evaluate the rationalize about the behavior and the performance of the learning algorithm. The three case studies that we undertake deal with:

1. Array shape, size and the mapping of a monolithic systolic array

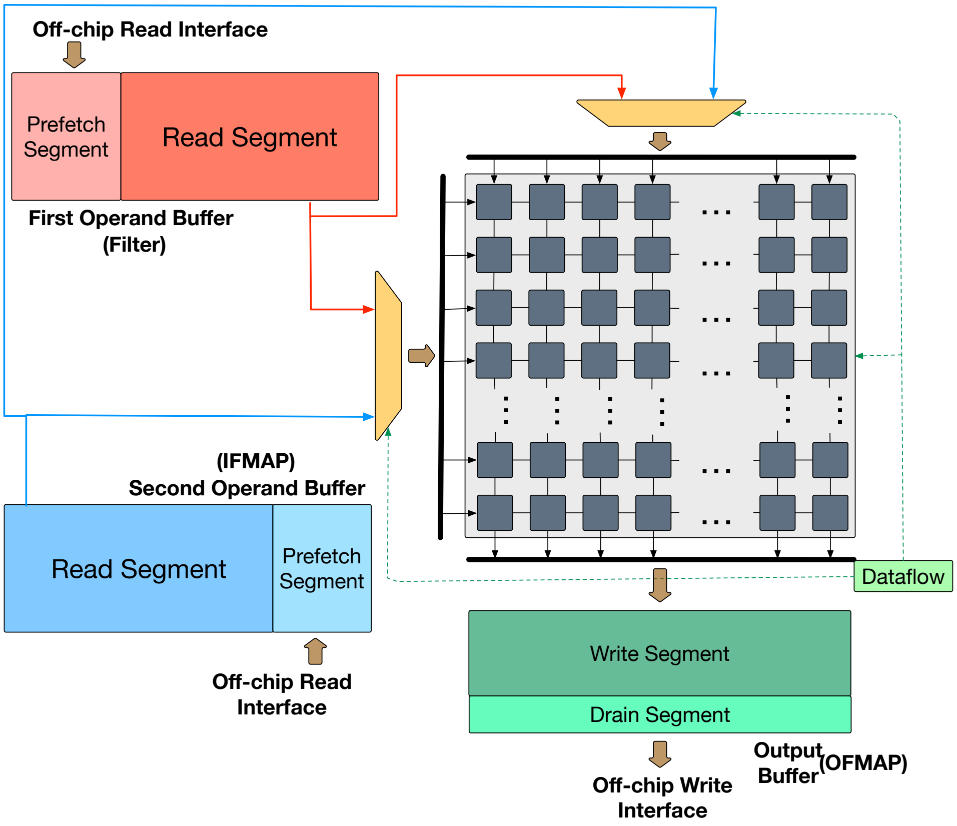


Figure 5.1: Monolithic systolic array template with configurable array dimensions, choice of dataflow, and configurable buffer sizes

2. The total size and the relative partitioning of the SRAM buffers in a monolithic systolic array
3. Mapping of various workloads onto different arrays in a heterogeneous systolic array setting.

We elaborate on each of the case studies in subsequent sections.

5.2.1 Case 1: Optimal Architecture and Mapping for Monolithic Systolic Array

Figure 5.1 depicts the schematic of a generic monolithic systolic arrays based accelerator. For the first case study we focus on designing the optimal compute array for a given workload. Recalling from our discussion in Chapter 3, extracting the most performance is not solely determined by allocating the more number of compute (multiply and accumulate

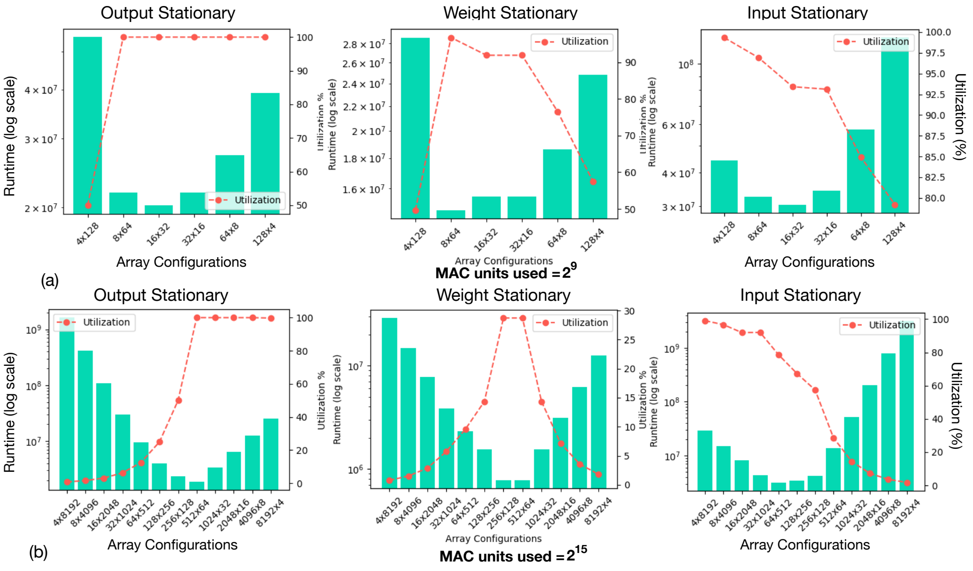


Figure 5.2: Variation of runtime and utilization (red dotted line) when using different array shapes and dataflows for (a) 2^9 MAC units, and (b) 2^{15} MAC units for first layer in ResNet18

(MAC)) units. For a given DNN layer, or a GEMM operation the shape of the computation needs to be in agreement with the mapping and the shape of the array to effectively use the given array. This point is further illustrated in by the charts depicted in Figure 5.2. In Figure 5.2(a) we show the runtimes and the array utilization in terms of mapping when when dataflows are employed while running the first layer of Resnet-18 [46] network. For each of the charts the x-axis denotes the various array configurations (row× columns) of the systolic array, all having the same number of MAC units (2^9). Figure 5.2(b) depicts similar information, but for the various array configurations with 2^{15} MAC units.

A couple of observations can be made from these charts. First, we note that although there is a correlation between high utilization and low runtime, the configuration with highest utilization does not translate to be the most performant configuration. Recalling our discussion from Chapter 3, several factors like the number of folds and serialization delay are also the factors that play additional role in determining the best performant configuration. Second, the best performant configuration and dataflow changes when the number of MAC units used changes, even for a the same workload. Same is the case for utilization. These variations are significant since these results demonstrate the huge

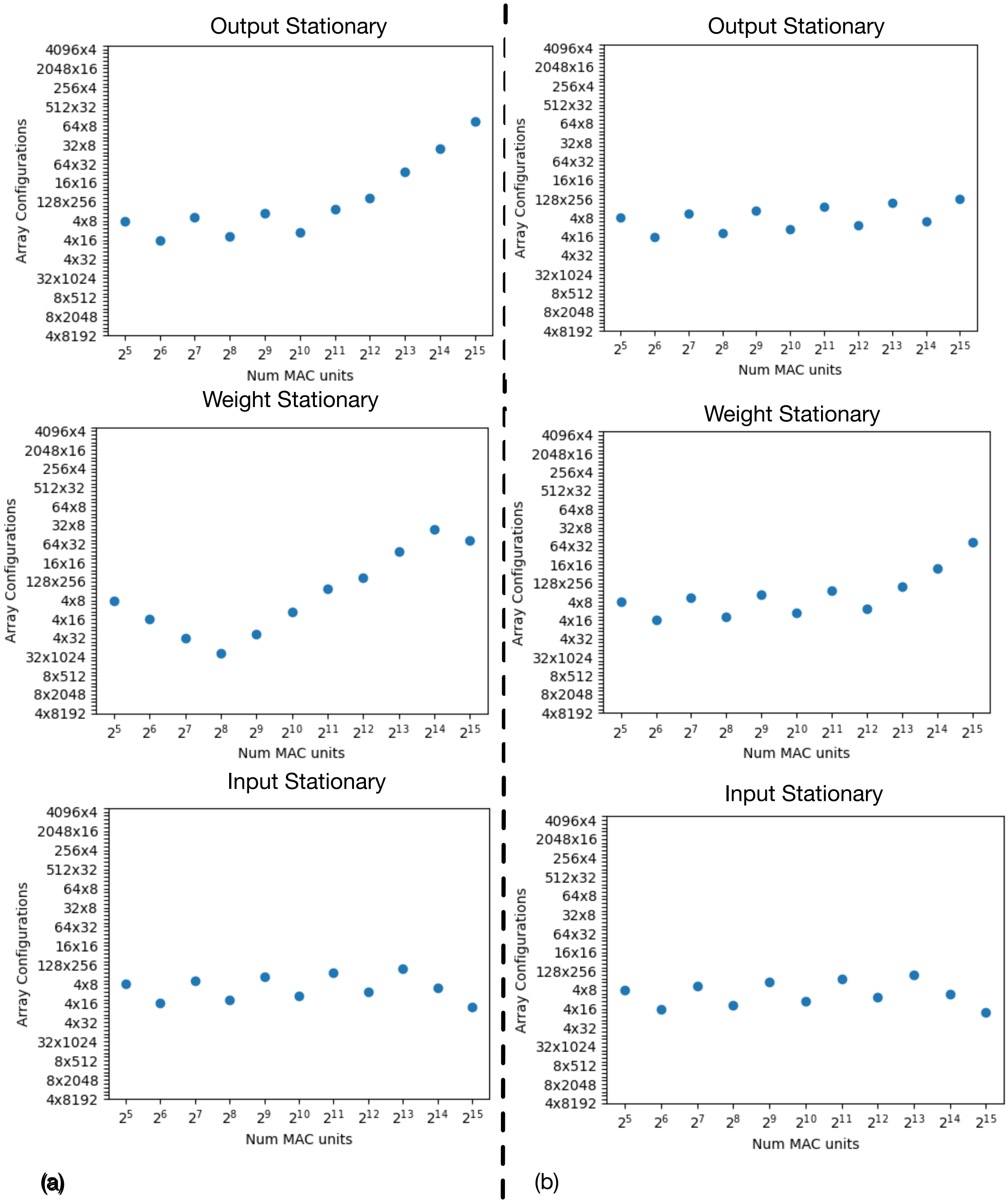


Figure 5.3: The variation of optimal array dimensions as the number of compute units vary from 2^5 to 2^{15} for (a) the first layer and (b) the eighth layer in Resnet18 network for different dataflows.

variation in the space of optima for just compute dimensions and mapping, even when the workload remains the same. Lack of any apparent pattern naturally is a problem for a learning algorithm targeted for this space.

As the optimal array configuration changes with the number of MAC units, it is interesting to determine if there is a pattern that the change follows. If such pattern exists then potentially a learnt model can use that to determine best configurations. Figure 5.3 depicts the trends in optimal array configurations for different dataflows as the number of MAC units are increased in exponents of 2. In each of this figure the y-axis represents array configurations sorted by aspect ratios (ratio of rows:columns) where the 'fat-short' configuration are placed towards zero and 'tall-skinny' configurations are placed near the top, putting the regular square configurations towards the middle of the axis. Figure 5.3(a) captures the variation of the optimal configuration for the first layer in Resnet18 network. A first order analysis does reveal some patterns, for instance the optimal configurations for input stationary dataflow appear to be localized with small rows and relatively wider columns, while the configurations for weight stationary dataflow seems to favour short-wide configurations for fewer MAC units and then start favouring configurations with more rows then columns. Similar changes in 'preferences' can also be observed for favourable configurations using output stationary dataflow. Taking the example of yet another layer in the network (eighth layer), chosen randomly, we observe similar trends as depicted in Figure 5.3(b). However, the takeaway from these charts is that further investigation is required to ascertain that the patterns generalized and therefore are relevant. Furthermore even if they are relevant, the seemingly arbitrary trends foreshadow that simple models with low dimensionality are unlikely to faithfully capture the landscape of optimal points. We therefore need to create a dataset with large number of points and perform systematic model exploration to learn the design space of optimal points.

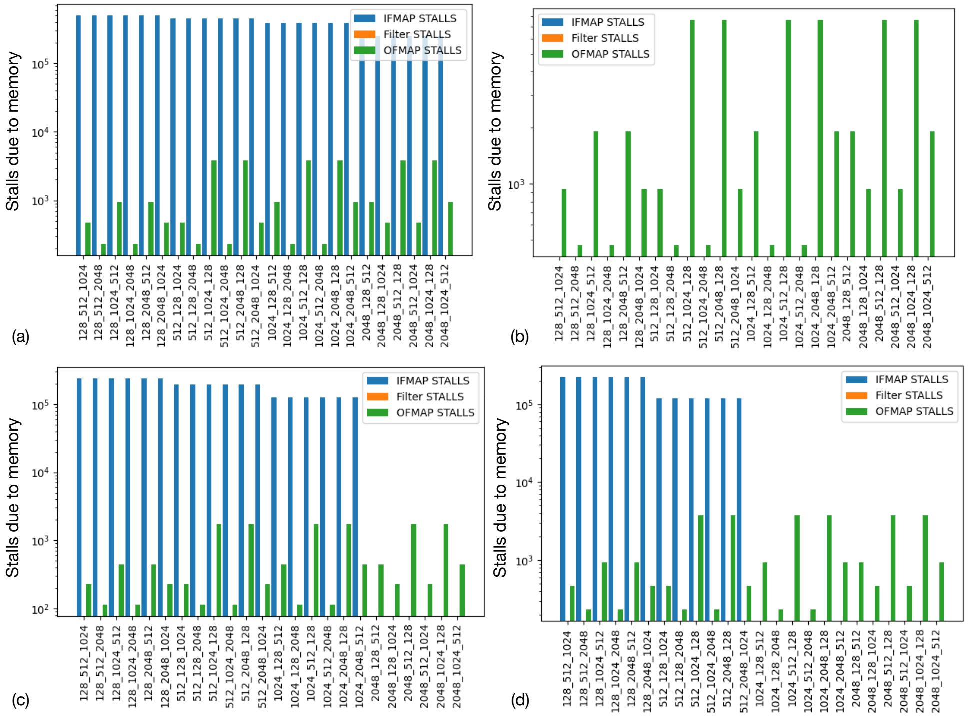


Figure 5.4: Variation of stalls encountered due to memory limitation vs memory size categorized per buffer type for GoogleNet's[54] second layer when using (a) OS dataflow, 32x32 array, and interface BW of 50 bytes per cycle; (b) IS dataflow, 32x32 array, 50 bytes/cycle BW (c) OS dataflow, 32x32 array, 100 bytes/cycle BW (d) OS dataflow, 8x128 array, 50 bytes/cycle BW

5.2.2 Case 2: Optimal Memory Partitioning for Monolithic Systolic Array

Proper memory sizing is yet another important parameter which determines both the performance and energy efficiency of an accelerator. Figure 5.1 depicts the template of the on chip SRAM memories employed in a systolic array based accelerator design. For GEMM operations, buffering is required for two operand matrices and the elements of the generated output before they can be drained off-chip. These buffers can be implemented either as logical sections of a monolithic memory bank or as physically distinct structures as depicted in Figure 5.1. The figure denotes the distinct memories in using the terminology of CNN matrices. The two buffers for hosting the elements of the input matrices are denoted as Input Feature Map (IFMAP) and Filter buffers, while the one for hosting the elements of the output matrix is denoted as the Output Feature Map (OFMAP) buffer. These memory segments are in general implemented as double buffered storage to exploit reuse and absorb the bandwidth mismatch of the interface links and the demand from the compute array. In double buffered implementation, some portion of the total memory size (typically 50%) is used to prefetch data from the external interface while the rest of the buffer is used to service the requests performed by the compute array. Once the elements in the read portion of the buffer are no longer required for computation, the prefetch segment is used for servicing request from the compute array while the rest of the memory is overwritten with the prefetched data.

During the design process however, real estate is allocated in terms of area which can be translated into total memory capacity. The task of the architect then is to determine the best allocation of this capacity into the three logical or physical buffers depending on implementation choice and the requirements of the workloads. The best design is of course the one which takes the minimum real estate in term of capacity but provides enough buffering such either there are no stalls arising due to buffering or at least the stall encountered are minimized within the determined budget.

Similar to the previous case study, we are interested in determining if there are any learnable patterns present in optimal memory sizing when running DNN workloads. From traditional knowledge of memory design we know that optimal buffering primarily depends on the dimensions of the array, the workload dimensions, mapping in terms of dataflow, and the bandwidth of the external interface supporting the buffer. To understand the cost landscape of buffer design, we take the example of the second layer in GoogleNet [54] in Figure 5.4. In each of these figure the x axis depicts the different memory configurations, broken down in terms of sizes of each buffer in KB. For example, the configuration 128_512_1024 denotes a configuration where the IFMAP buffer is 128KB of capacity, Filter buffer is 512KB, and the OFMAP buffer is 1024KB worth of capacity. For keeping the search space simple yet representative of the problem, we allow each buffer size to have values in the set 128KB, 512KB, 1024KB, and 2048KB without repetition. The y axis depicts the stalls encountered in serving the read and write requests for operand and result matrices.

The overall cost of stalls encountered is measured as the maximum of the stall attained of the three matrices. The best configuration naturally is the one which encounters the minimum cost. For configurations with equal costs, the one which uses the least amount of capacity is deemed as optimal.

Figure 5.4(a) depicts the cost landscape of second layer in GoogleNet on a 32 × 32 array, using OS dataflow, with the interface bandwidth of 50 elements per cycle for each buffer. We observe that in this setting the IFMAP operand matrix encounters the most stalls and hence determines the performance. Naturally the configurations which allocate the most capacity to IFMAP buffer (2048KB) are optimal. Since, even with the most capacity, stalls encountered by IFMAP elements dominate, the best configuration is the one which uses 128 KB for Filter buffer and 512 KB for OFMAP buffer. The effect of changing the dataflow, can be observed in Figure 5.4(b). This figure depicts the cost landscape when the dataflow is changed to Weight Stationary (WS) with all other parameters remaining the same. We immediately notice that across all the configurations only the OFMAP accesses encounter stalls. Hence the optimization goal then shifts to allocating the maximum size to the OFMAP buffer, hence making the configurations 128_512_2048 and 512_128_2048 both as optimal configurations.

In Figure 5.4(c) we depict the effect of interface bandwidth on the cost landscape. For this case, we again plot the stalls obtained for the second layer in GoogleNet on a 32 × 32 array, using OS dataflow, but the interface bandwidth is now changed to 100 elements per cycle for each buffer. The immediate difference when compared to the case of Figure 5.4(a), when allocating 2048KB for IFMAP buffer, the stalls due to the IFMAP matrices are nullified and the optimal configuration is determined by OFMAP stalls. The figure shows that the configurations with 1024 KB allocated for OFMAP buffer lead to the same stalls, and out of the two equivalent configurations the one with the least size ie. 128KB Filter buffer turns out to be optimal. On a similar vein we show the affect of array dimensions in Figure 5.4(d). As compared to Figure 5.4(a), the only change in this case is that we use an 8 × 128 array which has the same number of MAC units as the original 32×32 array, with all the other parameters being the same. The dimension change reduces the pressure on the IFMAP buffer and again the OFMAP buffer determines the best configuration. From the figure we see that configurations with 2048KB allocated to OFMAP, and 1024 KB allocated to IFMAP proves to be least costly. Hence rendering the configuration point 1024_128_2048 as the optima.

The takeaway from this simple experiment and the associated discussion is that even for a simplistic use case input parameters drastically change the points of optima. It is clear that simple learning model is unlikely to capture the mapping from the input parameter space to the space of optima.

5.2.3 Case 3: Optimal Compute Scheduling for Multi-Array System

The previous case studies primarily focus on the determining the optimal architecture parameters for a systolic array based accelerator. In this case study we examine a pure mapping problem involving multiple accelerators. The problem at hand is to find out the mapping on workloads on a multi-accelerator setting. To keep the problem set tractable we chose number of workloads to be mapped at once is equal to the number of accelerators we have available Figure 5.5 provides more information about the problem. This figure depicts three distinct accelerators, one monolithic 32×32 systolic array, a coarse grained distributed systolic array based accelerator setup with 4 8×8 arrays, and a fine grained distributed accelerator with 256 2×2 arrays. The number of MAC units in all three accelerators are kept constant to ensure that the choice of mapping a workload on any array is purely on the basis of mapping and not on computational upper-hand. Three workloads can be mapped onto these three accelerators at a single instance. The task of the mapper (either manual or automated) is to find the best mapping strategy such that all the three computations can complete in minimum time. The best mapping strategy not only includes the mapping of each workload to the corresponding accelerator but also the

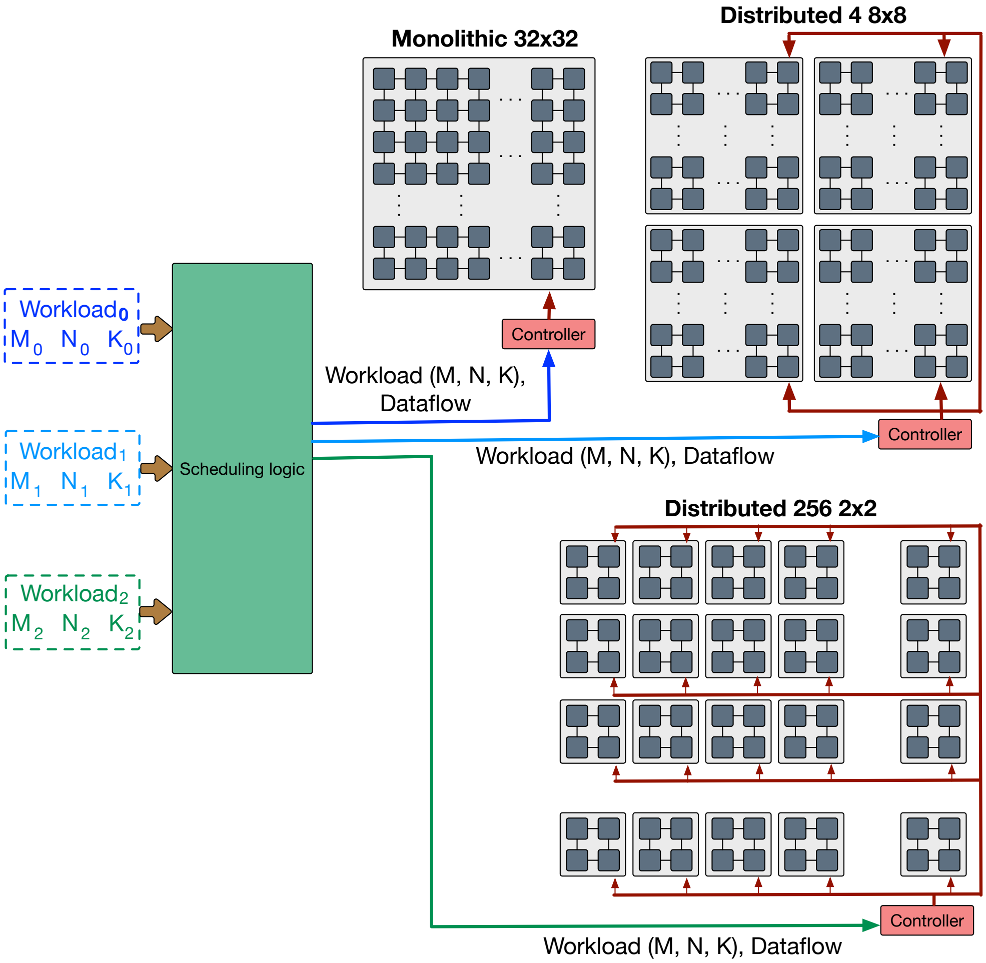
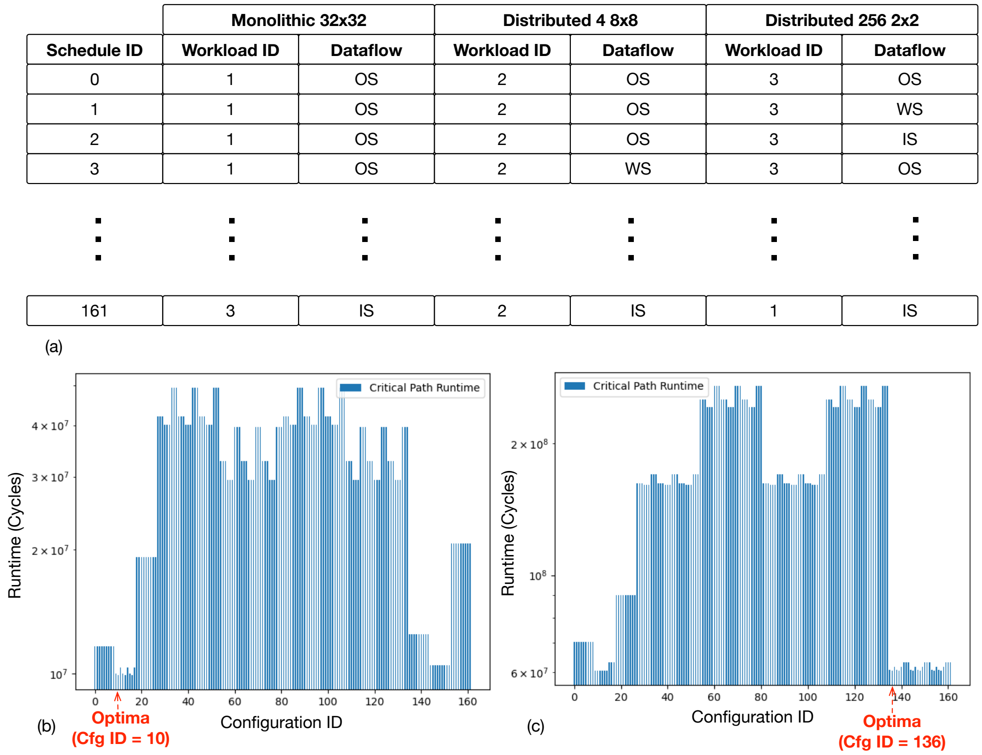


Figure 5.5: Schematic description of the multi-array scheduling case study

dataflow to be used.

Similar to the other two case studies, before we can analyse the feasibility of training a learning algorithm on this task, we take some examples to estimate the cost landscape for determining the optima for one example. In Figure 5.6(a) we show a scheme of indexing the various possible mapping strategy, that we refer to as a *'schedule'* for the three accelerator system that we show in Figure 5.5. Each entry in the table captures the workload ID and the chosen dataflow for the monolithic 32 × 32 array, the distributed 4 8 × 8 array, and the 256 2 × 2 arrays. For this example we see that there is a total of 162 different *schedules* possible. To understand the cost landscape we search through all the configurations and observe the cost in terms of critical path runtime for a couple of example workloads.



	Monolithic 32x32		Distributed 4 8x8		Distributed 256 2x2	
Schedule ID	Workload ID	Dataflow	Workload ID	Dataflow	Workload ID	Dataflow
0	1	OS	2	OS	3	OS
1	1	OS	2	OS	3	WS
2	1	OS	2	OS	3	IS
3	1	OS	2	WS	3	OS
⋮	⋮	⋮	⋮	⋮	⋮	⋮
161	3	IS	2	IS	1	IS

Figure 5.6: (a) Table of scheduler IDs and corresponding configuration (b) Runtimes of the critical path for the first layer in GoogleNet, YoloTiny, and Alexnet (c) Runtimes of the critical path for the third layer in GoogleNet, YoloTiny, and Alexnet

In Figure 5.6(b) we plot the critical path runtime for the first layer of three popular convolution neural networks, GoogleNet[54], YoloTiny[55], and Alexnet[56] respectively. For this particular layer we observe that, the *schedule* with indexed by ID 10 emerges as the optima. Interestingly when we observe the cost landscape for the third layers in the respective network, we observe that it is quite different from the case of first layers of the networks and an the configurations close to the optima lie with larger schedule IDs, with 136 being the best. The cost landscape for each layer themselves do not depict any particular pattern that could be learnt to train a predictor since (a) it is not uniform and (b) the nature of the landscape is heavily dependent on the workload parameters.

In the next section, we therefore turn our attention to the space of just the optimization points for the three workloads instead of the cost landscape to identify learnable patterns.

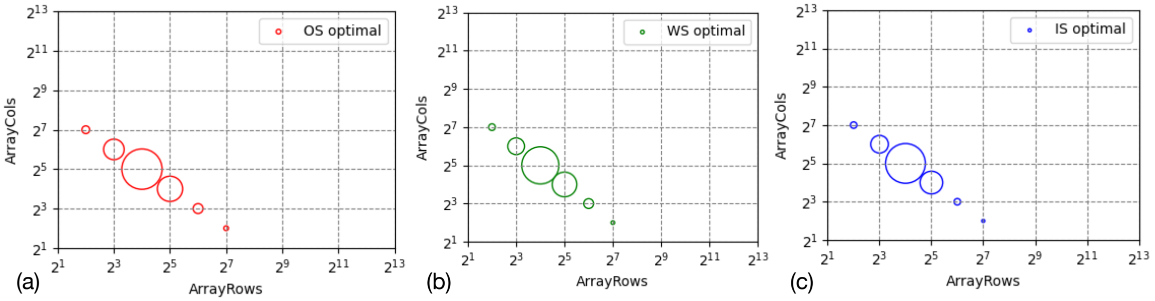


Figure 5.7: Relative frequency of optimal array dimensions obtained for GEMM workloads using (a) Output Stationary, (b) Weight Stationary, and (c) Input Stationary dataflows

5.3 Design Aware Analysis

In the search of learnable patterns for determining the optimal parameters of systolic array based accelerator design, we turn our attention to variation of the optimal parameters themselves. In this section we examine the optimal parameters and their correlation with the input and other system level parameters. The following subsections provide evidence of learnable patterns for the first to case studies involving array configuration, dataflow, and memory design tasks.

5.3.1 Array Shape and Dataflow

First we take a look at the space of optimal array dimensions. We generated a dataset of 10^5 points, each comprising a GEMM workload represented by M, N, and K dimensions, and design constraint represented as maximum MAC units, and the best array configuration and optimal dataflow among OS, WS, or IS. The M,N, and K values are obtained by randomly choosing an integer value from a uniform distribution. The maximum mac unit is chosen as a power of two. To determine the best array configuration and dataflow, we search through all the possible dataflows, row and columns with MAC units less than or equal to the maximum MAC units specified.

Interesting patterns begin to emerge when the distribution of the optimal configurations are considered. For example, consider Figure 5.7(a). This is a scatter plots of all the optimal

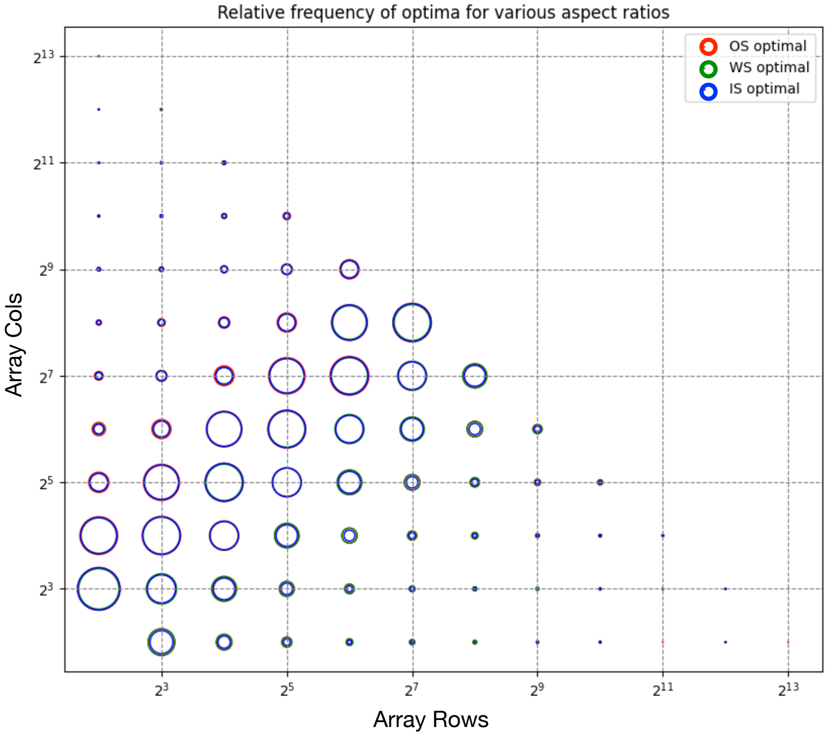


Figure 5.8: The pattern of optimal aspect ratio and optimal dataflow obtained for GEMM workloads on systolic arrays with 2^8 to 2^{15} MACs.

rows and columns which emerge as optimal for 10^4 data points for which the maximum MAC units is kept fixed as 2^9 and the optimal dataflow obtained is output stationary (OS). The radius of the marker in for each point captures the relative frequency. Figure 5.7(b-c) plots the same distribution of optimal points, but for the ones for which the optimal dataflow is determined to be weight stationary (WS) and input stationary (IS) respectively. A couple of observations are immediate from these charts. *First*, we observe the optimal array configurations all utilize the maximum MAC units. This is understandable since the configurations that would not follow this trend would only for the workloads which encounter severe under utilization for all array configurations with the maximum MAC units 2^9 which usually never happens. *Second*, we notice that the array configuration with almost equal number of rows and cols are favoured most for all the three dataflows.

Figure 5.8 shows the complete picture across the entire search space of this specific case

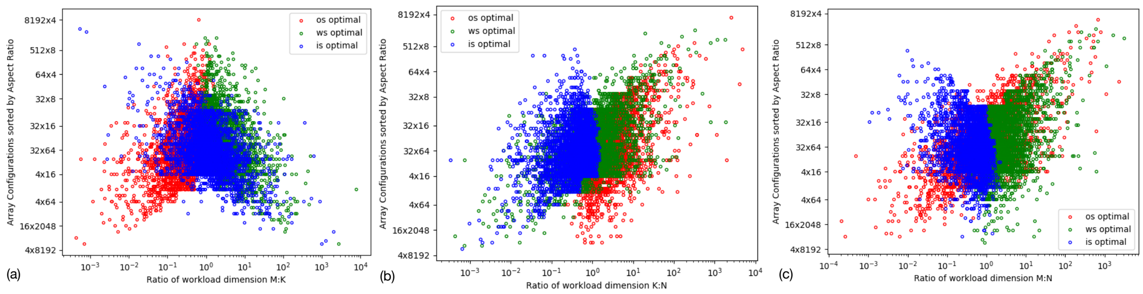


Figure 5.9: The correlation of optimal array dimension obtained (y-axis) and the matrix shape in terms of aspect ratio (x-axis) of (a) Input operand matrix (M×K) (IFMAP), (b) input operand matrix (K×N) (Filter), and (c) Output matrix (M×N) (OFMAP). The colors of the markers indicate the optimal dataflow obtained, highlighting the pattern in the design space1.

study This plot is a combination of optimum array configurations and dataflows as shown in Figure 5.7, but captures the points where the maximum MAC units are varied from 2^8 to 2^{15}, furthermore the array configurations corresponding to the different optimal dataflows are also consolidated in this charts and are plotted as markers with different colors. For instance the configurations with OS as the optimal dataflow are plotted as red markers, the ones with WS as optimal dataflow are plotted in green, and ones with IS as optimal are plotted as blue. With this chart, we can make observe some interesting patterns apparent in the distribution of the optimal configurations for a wide variety of input.

The *first* and immediate observation is about the relative frequencies of array configurations. The nearly square configurations with 2× the columns as the rows are optimal for most of the workloads, irrespective of the MAC unit constraints. The next two the most popular configuration are the ones with equal sized rows and cols, the configurations with 4× as many columns as rows, and the ones with 2× more rows as columns. As a simple rule of thumb a search among these four aspect ration variations might be sufficient to find the best array configuration for many workloads. However, there are two other subtle observations which are critical to build a robust learning model. The chart also shows that there are no points no valid points on the chart which does not serve as the optima for at least one data point. This shows that a higher dimensional model is required to faithfully capture the space of the optima. One more observation is about the correlation of the optimal dataflows and optimal dataflows. For most of the points, especially the ones which turn out to be optimal for many inputs, no reasonable conclusions can be made about weather any dataflow is preferred over another.

From our analysis in a Chapter 3 we know that the shape of the operands are one of the major factors that determine the optimal dataflow for GEMM operations on systolic arrays. Acting on this piece of knowledge we examine the optimal dataflows obtained and the shape of the operand matrices in Figure 5.9. In each of the plots in this figure, the x-axis plots the shape of an operand matrix in terms of increasing aspect ratio (the ratio of rows to columns); thus larger the value of x is the taller and skinnier the operand matrix it denotes. The y axis captures the various aspect ratios of systolic arrays ranging from ones with 2^8 to 2^{15} MAC units also sorted in terms of aspect ratios. Each point in any of the scatter plots depicts the operand shape for the GEMM operation, and the corresponding array configuration that comes out as the optimum. The color of the optimal dataflow attained is depicted using different colors.

These charts show telltale signs of the how the optimal aspect ratios can be determined by using the operand shape and optimal array configuration information. Figure 5.9(a) which captures the optimal array dimension and dataflows with the shape of the $M \times K$ GEMM or the IFMAP matrix shows a clear separation of output stationary (OS) vs weight stationary (WS) favouring configurations. However the information captured in this chart cannot be used to determine if input stationary would be optimal as compared to the other two alternative dataflows. The next chart Figure 5.9(b), capturing the filter operands or the $K \times N$ operand matrix shows a clear separation of the OS optimal and IS optimal cases, although WS optimal cases could not be differentiated among the other two. However, when considering a similar plot with output matrix shape in Figure 5.9(c) a clear distinction among IS and WS optimal cases is apparent. These three charts in conjunction depict that a statistical model should be able to be trained to predict the optimal dataflow for GEMM workloads mapped onto systolic arrays.

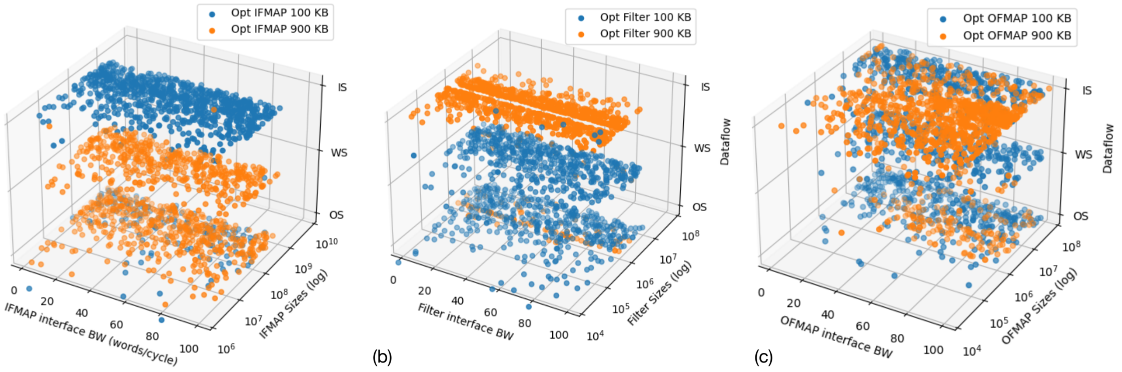


Figure 5.10: The relation of optimal buffer sizes of (a) IFMAP operand buffer, (b) Filter operand buffer, and (c) OFMAP buffer, with operand sizes, interface bandwidth, and dataflow.

5.3.2 SRAM buffer Sizing

We also examine the space of optimal memory configurations to find out the patterns that could be leveraged for learning the design space. Similar to the case for optimal array dimension and dataflow above, we perform a design aware analysis by visualizing the optimal sizes obtained for each buffers for about 10^4 data points.

Each data point in our analysis is obtained as following. First we chose the GEMM workload dimensional by randomly sampling integers as explained in the case study above. We also randomly sample the rows, columns from a set of powers of two such that the total number of MAC units is $< 2^{20}$. A random integer < 100, is sampled from an uniform distribution to denote the interface bandwidths of the buffers. We also allocate a maximum memory constraint by randomly choosing a number between 300KB to 3MB. Next multiple memory configurations are evaluated for the obtained workload and compute system configuration to determine the memory sizes which lead to least amount of stall while taking the least possible buffer capacity.

Figure 5.10 plots the trends observed for the optimal memory sizes thus obtained with respect to the interface bandwidth, the size of the matrices, and the dataflow used. For simplicity of plotting we show only two sizes. The small 100KB buffer size represented by blue dots, and large 90KB buffer size represented by orange dots. Figure 5.10(a) depicts the distribution of optimal IFMAP buffer sizes. We notice that the main determining factor in this case is the dataflow. The output and weight stationary dataflows predominantly prefer larger buffer sizes, while input stationary dataflow prefer the smaller sizes. This pattern is explainable by considering the fact that IS dataflow by its very nature maximizes the reuse in the IFMAP matrix, thus relieving pressure on the interface which translates to lower buffer requirements. On the other hand, the other two dataflows stream the elements of this particular matrix throughput the computation, which justifies the preference for larger buffer sizes.

5.4 Learning Architecture and Mapping Space

Our analysis in the previous section shows that the systolic-array-based accelerator design and mapping space possesses high-level patterns that indicate that it is perhaps possible to predict design parameters given that the distribution of the data is internalized. In this section, we discuss systematically structuring the learning problem, dataset generation, and briefly perform statistical analysis on the generated datasets.

5.4.1 Design Optimization as Learning Problem

The first step towards learning the observed patterns is to formulate the search-based optimization problems involved in our case study to machine learning-based regression or classification problem setting. Empirically we found classification tends to be a better fit for learning architecture and mapping optimization. A naive approach for employing classification for predicting architecture parameters is to independently predict each design parameter. However, there are a couple of problems with this approach. First, a separate model needs to be trained for each design parameter, which can get easily get out of hand when dealing with a system with significant complexity. Second, the parameters are often inter-dependent, for example, the sizes of memory buffers in the systolic array, and different models might fail to capture the interdependence if trained separately. Motivated

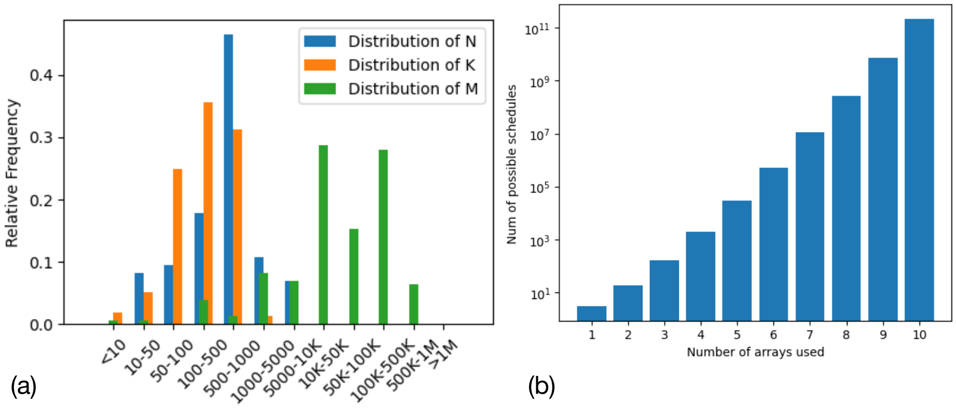
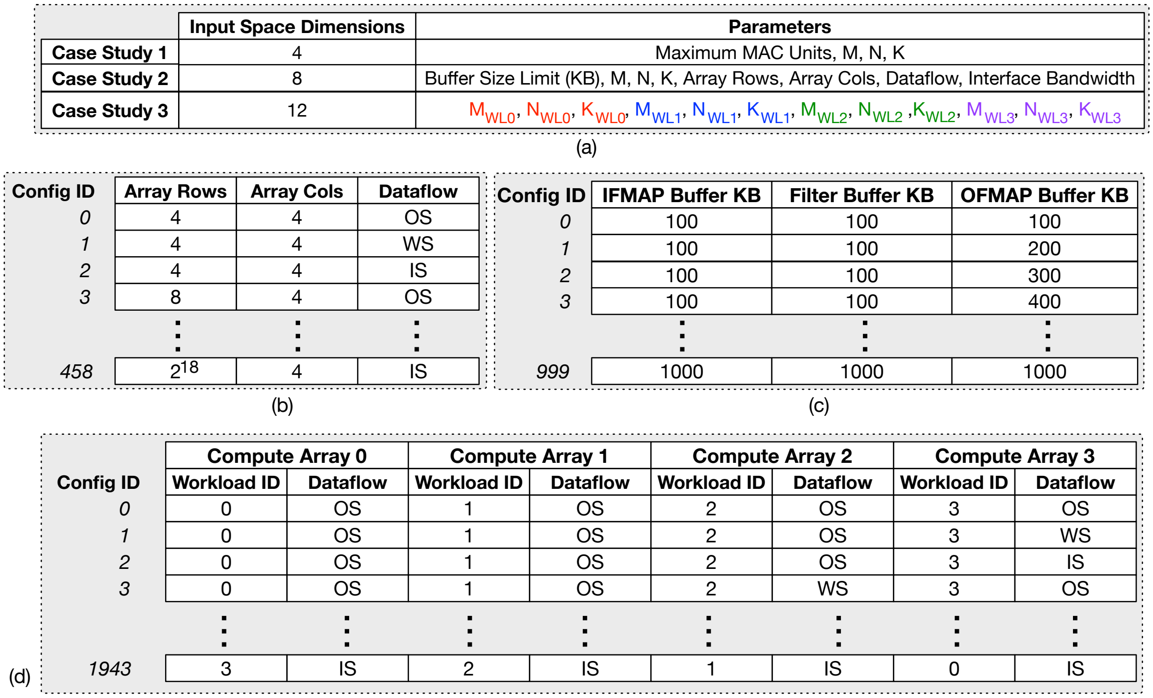


Figure 5.11: (a) Chart showing the distribution of operand matrix dimensions for GEMM operations involved in layers of popular neural networks (b) Growth of the scheduling space

by these factors, we formulate the problem into a recommendation setting, where a bunch of parameters is clubbed into bins, and the trained model is expected to recommend the bin corresponding to the optimal parameters. This formulation helps us leverage the existing classifier models developed by the machine learning community and also helps us reason about the problem systematically.

5.4.2 Dataset Generation

Case Study 1: Array and Dataflow prediction. For this case study, we want to predict the optimal dimensions of a systolic array and the dataflow, given then design constraints and the dimensions of the GEMM workload. *The input space* our for this task, therefore comprises of four integers, three for the operand dimensions (M, N, and K), and one integer capturing the maximum number of MAC units allowed. To keep the input space bounded, the MAC units are provided in exponents of 2. We limit the maximum possible MAC units at 2^{18}. The workload dimensions are provided as randomly sampled integers from a uniform distribution. The limits of the workload dimensions are determined by the distribution depicted in Figure 5.11(a). This distribution is generated from the layer dimensions of popular convolution neural networks [56, 57, 54, 46] . This distribution dictates that the values for M dimension vary between 1 to 10^5, N in [1, 10^4], and K in



	Input Space Dimensions	Parameters
Case Study 1	4	Maximum MAC Units, M, N, K
Case Study 2	8	Buffer Size Limit (KB), M, N, K, Array Rows, Array Cols, Dataflow, Interface Bandwidth
Case Study 3	12	M_{WL0}, N_{WL0}, K_{WL0}, M_{WL1}, N_{WL1}, K_{WL1}, M_{WL2}, N_{WL2} ,K_{WL2}, M_{WL3}, N_{WL3}, K_{WL3}

(a)

Config ID	Array Rows	Array Cols	Dataflow
0	4	4	OS
1	4	4	WS
2	4	4	IS
3	8	4	OS
	⋮	⋮	⋮
458	2^{18}	4	IS

(b)

Config ID	IFMAP Buffer KB	Filter Buffer KB	OFMAP Buffer KB
0	100	100	100
1	100	100	200
2	100	100	300
3	100	100	400
	⋮	⋮	⋮
999	1000	1000	1000

(c)

	Compute Array 0		Compute Array 1		Compute Array 2		Compute Array 3	
Config ID	Workload ID	Dataflow	Workload ID	Dataflow	Workload ID	Dataflow	Workload ID	Dataflow
0	0	OS	1	OS	2	OS	3	OS
1	0	OS	1	OS	2	OS	3	WS
2	0	OS	1	OS	2	OS	3	IS
3	0	OS	1	OS	2	WS	3	OS
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1943	3	IS	2	IS	1	IS	0	IS

(d)

Figure 5.12: (a) Input space size and input parameters for the three case studies; Output space of (b) Systolic Array dimension and dataflow prediction case study (c) Memory buffer size prediction case study (d) Distributed array schedule prediction case study

[1, 10^3]. *The output space* is a list of labels for this task. As shown in Figure 5.12(b) each label serves as the index for a set of parameters, which for this case study denote the systolic array height, width, and one of the dataflows. Keeping with the conventional systolic array designs, we only consider the array dimensions that are powers of 2. In our experiments, we allow the minimum array dimensions to be 2^2 while the maximum dimensions, dictated by the maximum possible MAC units in the input space, come out to be 2^{18}. With these limits in place, the output space for our problem contains 459 different array and mapping configurations. To generate the dataset, we use runtime as the cost metric. We use the SCALE-Sim [58] simulator to generate runtimes for each workload for a given array dimension and workload. We modify the simulator to generate only compute runtime and ignore stalls, speeding up the search. We exhaustively search through all the valid configurations to generate the label. We generated about 5 million data points, which took about a couple of weeks of times when running over several servers using a total of

800 logical CPU cores.

Case Study 2: Buffer Size prediction. In the second case study, our goal is to predict all optimum sizes of the three operand buffers in the systolic array (see Figure 5.1) simultaneously for given workload parameters, information about the compute, and design constraints.

The input space, as depicted in Figure 5.12(a), captures the maximum memory capacity, workload dimensions (M, N, K), the array dimensions, dataflow, and interface bandwidth of the buffers. For simplicity, we assume the same interface bandwidth for all three buffers. The limit for the maximum memory capacity is set to 3000 KB; the workload dimensions are sampled from the same distribution as the previous case study with limits observed in Figure 5.11(a). The array dimensions and dataflow are sampled from the output space of the previous case study to keep our experiments consistent. The interface bandwidth expressed in bytes/cycles is taken from the space of integers in [1, 100], sampled with uniform probability. *The output space,* is a list of labels, where each label indexes an entry of memory sizes for each of the buffers. The minimum size of each buffer is restricted to 100 KB, and the allowable sizes increment with a step size of 100 KB as well. The maximum allowable size is 1 MB for an individual buffer. With these constraints, the output space contains 1000 distinct points defining the search space. For this case study, the optimal memory size is the one that leads to a minimum or no stall. For different configurations with the same stalls encountered, the one with the smallest cumulative capacity is considered the best. For this case study as well, we use SCALE-Sim[59] to generate costs of the various memory sizes. We generate about 5 million data points, where a million point takes about a week when multiple runs are launched onto a server cluster totaling 800 logical CPUs.

Case Study 3: Multi-Array Scheduling. In Section 5.2.3 we describe the problem of scheduling a set of GEMM workloads on an equal number of a distinct heterogeneous collection of compute units. *The input space*, for this problem, is simply the GEMM workload dimensions (M, N, K) one for each compute array. *The output space*, is an id depicting the mapping of workload to the arrays and the corresponding dataflow to be used as shown in Figure 5.6(a).

It is worth noting that our case study concerns the schedules when the number and the dimensions of the arrays are fixed. This case study does not cover the cases where the number of arrays or the workloads changes. However, it is interesting to observe that when the number of arrays changes the possible number of scheduling strategies grows exponentially as well as combinatorially as depicted by the equation $N = 3^x \times x!$, where x is the number of compute arrays.

Figure 5.11(b) depicts the growth of the space with the number of compute units. *In this case study, we chose to learn the scheduling space of* ***four*** *arrays, which leads to 1944 possible entries in the output space.* The decision to chose this configuration is purely pragmatic, intended to keep the dataset generation times bounded. The different arrays used in the study is shown in Figure 5.12(c).

Run time is used as the cost function for dataset generation. The schedule which leads to the least runtime of the slowest running workload is deemed to be the winner. For the schedules which have the same critical path runtime, the one with the least cumulative runtime is chosen to be optimal. We create an in-house simulator similar to [59] to compute the runtime of GEMM workloads on a distributed systolic array setting. Similar to the previous case studies we generate roughly 5 million data points for learning.

5.4.3 Statistical analysis

In Section 5.3 we observed that the optimal configurations show distinct patterns when analyzed with manually picked parameters. In this section, we perform statistical analysis on the generated datasets to gain additional insights on the ability to learn the distribution.

First, we analyze the distribution of the categories in the training dataset. In Figure 5.13(a-c) we plot the relative frequencies of the label categories for three case studies.

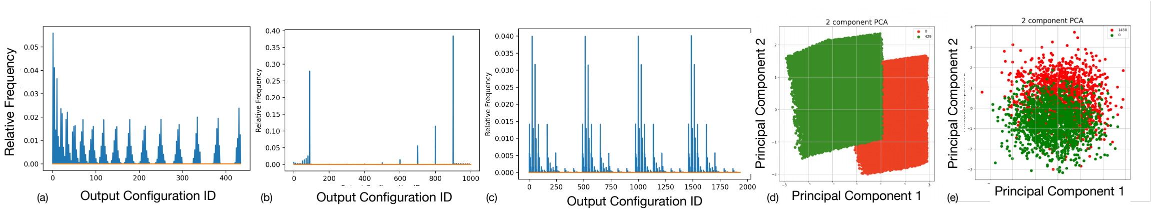


Figure 5.13: Distribution of the configuration ids for (a) Case Study 1, (b) Case Study 2, (c) Case Study 3. Distribution of data points along the most relevant principal component of two configurations for (d) Case Study 1, and (e) Case Study 3.

The dataset for our first case study (see Figure 5.13(a)), which captures the optimal array dimensions and dataflow for given GEMM workloads, we immediately notice that the output distribution is complicated and non-uniform, but structured. We observe that several configurations are represented in the majority while there exist a reasonable amount of configurations that have minuscule representation. This is consistent with the observations we make from Figure 5.8, where we see that highly skewed array dimensions are optimal only for a small number of use cases, leading to lower frequencies. It should be noted however that the input space of the dataset, which comprises the dimensions of the GEMM workload and the max number of MAC units is covered uniformly.

In Figure 5.13(b) we show the distribution of categories for the second use case, pertaining to the memory sizing problem. The immediate observation that can be made on this distribution is that for this problem, a handful of configurations work for a wide variety of input configurations, evident by the few classes the dominate the output distribution. The most frequent category accounts for 38% of the output. The two most frequent capture 66%. Continuing, the ten most frequent configurations account for roughly 90% of the output. Beyond this point, other configurations have very few occurrences and are therefore much harder to predict. We expect low prediction accuracy on this dataset due to the high bias in the dataset.

A similar analysis on the scheduling dataset, depicted in Figure 5.13(c) reveals a different pattern. In this case, not only there are a few very prominent configurations, but also a large number of categories with very low frequency. The vast majority of configurations

SVC RBF	Support Vector Classification using Radial Basis Kernel
SVC Linear	Support Vector Classification using Linear Kernel
XGBoost	eXtreme Gradient Boosting based Classifier
MLP-A	Multi-layer perceptron with 1 hidden layer with 128 nodes
MLP-B	Multi-layer perceptron with 1 hidden layer with 256 nodes
MLP-C	MLP with 2 hidden layer with 128 nodes each
MLP-D	MLP with 2 hidden layer with 256 nodes each
AIrchitect	Custom network shown in Fig1, with 256 hidden nodes

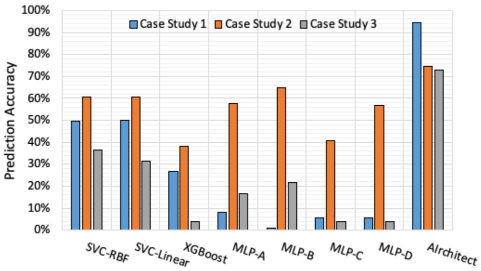


Figure 5.14: Performance of Classifier Frameworks

are never optimal and do not occur. We can see that four classes, in particular, dominate the spectrum. Each accounts for roughly 14% of the output totaling 56%. There are eight additional classes that each account for roughly 3% each totaling 24%. The twelve major classes then account for 80% of the optimal configurations. We can take away two key insights from this plot. First, a relatively simple model, which can classify the larger spikes can still provide respectable prediction performance. However, employing a more sophisticated model can significantly boost the performance of the schedule predictor, if it can learn to classify the labels with smaller frequencies.

Finally, we examine the efficacy of the handcrafted features used in the datasets. In Figure 5.13(d-e) we plot a subset of the data points along with the most prominent principal components for the first and the third case study respectively for two randomly chosen labels/categories. The visual separation of the points corresponding to different labels suggest that performing higher dimensional transformation with the chosen input parameters helps to choose a hyper-plane capable of classifying the data points. This chart provides a quality assurance that the datasets are classifiable and as a consequence can be learned, although not formal proof.

5.4.4 Learning with Existing Classifiers

The problem formulation discussed in Section 5.4, allows us to use off-the-shelf classifiers to capture the design space and make predictions about the optimal parameters for given workloads and design constraints. We test out various models with different degrees of complexity on the datasets generated for the three case studies. The table in Figure 5.14 show these models. To summarize we used the scikit-learn [60] libraries implementation of support vector classifiers [61] with linear and radial basis kernel. The state-of-the-art tree boosting method called eXtreme gradient boosting (XGBoost [62]) available from the xgboost package [63]. We also implemented four multi-layer perceptron (MLP) networks in TensorFlow's Keras [64, 65].

Figure 5.14 shows the accuracy obtained for the three case studies when 2×10^6 data points are used for fit/training. For the MLPs, the networks are trained for 15 epochs with 90:10 training-validation split. We used a categorical cross-entropy loss function with accuracy as the optimization objective. Among the various case studies, the one for memory size prediction is learned well among all the models, with MLP-B attaining about 63% validation accuracy. Support Vector Classification, however, was able to perform the best among the models considered attaining about 50% test accuracy for case study 1, 60% and 40% for the other two respectively.

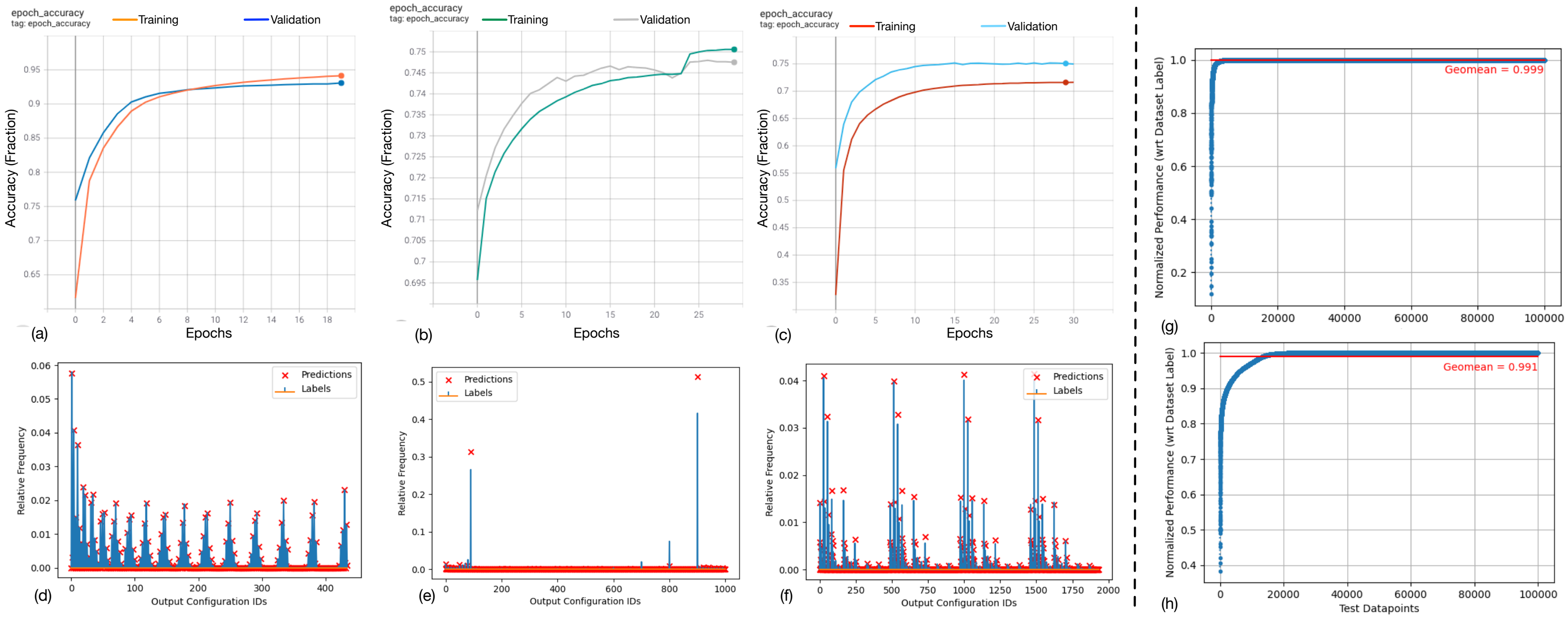


Figure 5.15: Progression of training and validation accuracy vs epochs when training AIrchitecton the dataset for (a) Case Study 1, (b) Case Study 2, (c) Case Study 3. The distribution of actual label and predicted configuration IDs by trained AIrchitect on test datasets with 10^5 points for (d) Case Study 1, (e) Case Study 2, (f) Case Study 3. The sorted normalized performance of predicted configurations to the optimal configurations on workloads present in test dataset for (g) Case Study 1, (h) Case Study 3

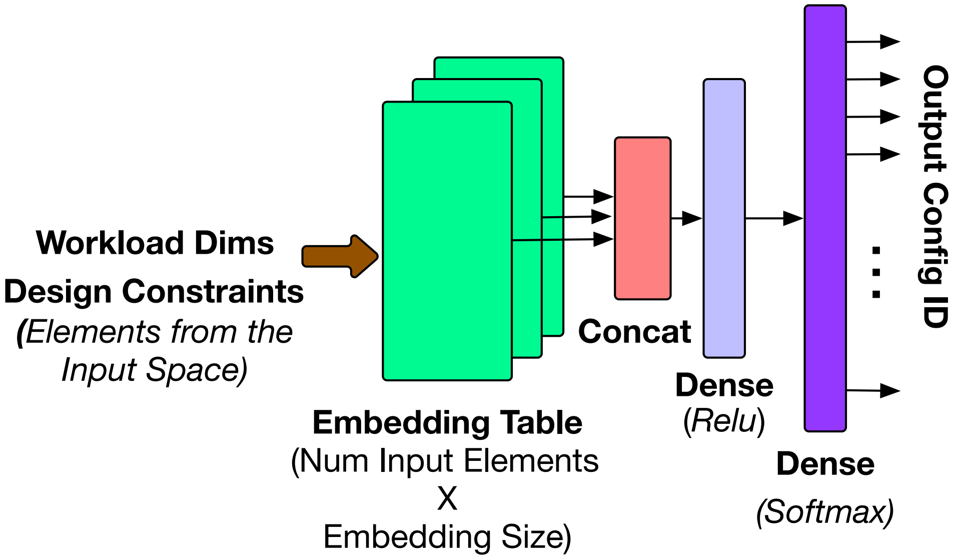


Figure 5.16: Generic structure of the custom designed recommendation network

5.5 AIrchitect: Design and Analysis

From our analysis in the previous section Section 5.4.4, we notice that although off-the-shelf classifiers are capable of learning the design space for our various use cases no single model appears to perform consistently across the different spaces. We design a general network structure, which we call AIrchitect, intending to obtain the capability to achieve reasonable learning performance across different distributions. Figure 5.16 depicts the general structure of our proposed model. The inspiration for this design comes from the structure of the modern recommendation networks like DLRM [66], which perform recommendation by mapping query inputs onto a trained embedding space, followed by MLP based classification. The trained embedding map the input data from the user-defined input space onto a latent space, which immensely improves the performance of the classifiers. This is also evident from the performance of MLP-B vs AIrchitect as depicted in Figure 5.14. Among the various case studies, the number of inputs and outputs are the parameters that we changed. The number of inputs to the network is equal to the input space, while the network generates a one-hot vector of length equal to the output space of the problem indicating the optimal design or mapping parameters, as discussed in

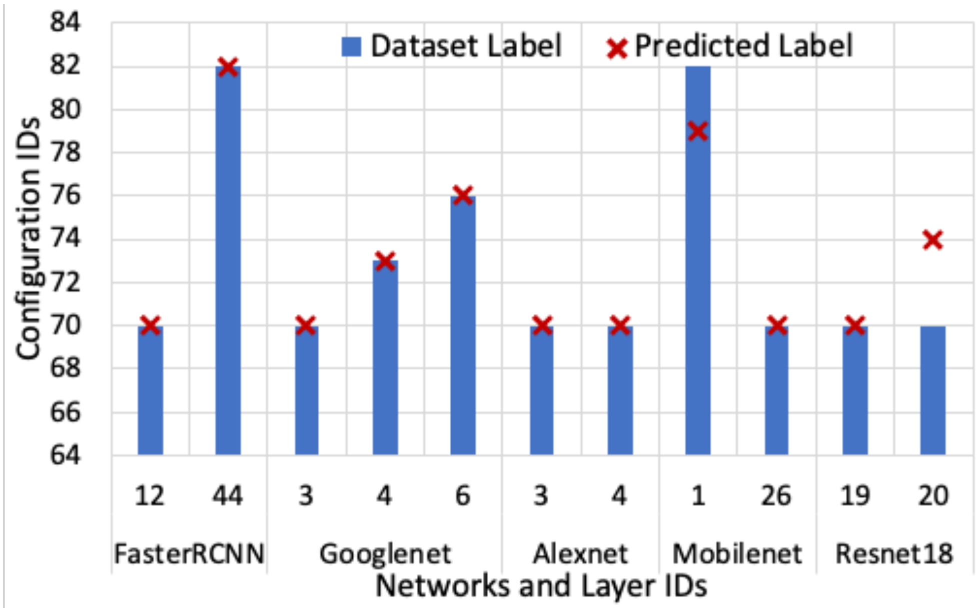


Figure 5.17: Predicted and actual labels for case study 1 for a few layers of popular CNNs

Section 5.4.2, Figure 5.12. We use an embedding size of 16 and a 256-node MLP hidden layer for our models across the case studies.

For all three use cases we train the corresponding versions of AIrchitect on the respective datasets with 4.5M points (Section 5.4 using 90:10:10 train-validation-test split). We use TensorFlow's Keras library[64, 65] to implement the network and train using categorical-cross-entropy as the loss function, with accuracy as the optimization metric. Figure 5.15(a-c) shows the accuracy obtained during training for the three case studies respectively. We observe that the design space of case study 1 is learned with a high validation accuracy ($> 94\%$) in about 15 epochs. The training for case study 2 finishes, in about 22 epochs achieving a validation accuracy of 74% before starting to overfit. For case study 3, the network saturates at about 15 epochs at about 76% validation accuracy. As we see in Figure 5.14, AIrchitect beats the best performing off-the-shelf classifiers at least by about 10% accuracy.

To gain further insights on the quality of training, we plot the distribution of the actual labels and the predicted configuration IDs for 10^5 previously unseen test data points for each case study. Figure 5.15(d-f) shows predicted vs actual distribution for the three case studies. The first observation is that the test datasets' actual distribution closely matches with distributions of the original dataset, shown in Figure 5.13(a-c) corroborating that the test set does not compromise on generality. Second, visually the predicted distribution for case study 1 almost perfectly matches the actual distribution, confirming that the network learned the design space. Third, we observe that the networks for case study 2 and 3, learn to predict the configurations with significant representation on the dataset, while the configuration IDs with low statistical probability is ignored as noise. The presence of large spikes in the case of memory size prediction biases the model, shown by the high frequencies of the top two configuration IDs (Figure 5.15(e)) leading to relatively low accuracy. Similarly, for case study 3,Figure 5.15(f) shows that the model was successful in learning the distribution of the large spikes. However, in doing so it ignored the configurations with lower frequencies, which in turn cumulatively lowered the accuracy figure. *However, it is worth noting that the model ignoring the lower frequencies as statistical noise demonstrates that the model is robust and does not overfit for any of the use cases.* Further improving prediction accuracy for different design spaces requires data engineering on a case-by-case basis and therefore, is out of the scope of this paper.

To understand the cost of misprediction within the use cases, we compute the performance (reciprocal of runtime) of the workloads in the test dataset, for the predicted and label configurations. In Figure 5.15(g) we show the performance of the predicted configurations normalized to the labels for the 10^5 data points. Due to the high prediction accuracy, we observe that only a few data points have catastrophic performance losses (¡20% of the optimal) leading to a 99.99% average performance (Geometric Mean) of the best possible. In Figure 5.17 we depict the performance of the network on some layers from networks like FasterRCNN[67], GoogleNet[54], Alexnet[56], MobileNet[57] and ResNet-18[46]. The layers of any of these networks were not part of the training or validation dataset, but the model is able to predict the optimal array shapes and dataflow, when queried with a constraint of 2^{10} MAC unit limit. Interestingly for case study 3, which had relatively low accuracy, for most cases, the performance does not lead to catastrophic losses. Figure 5.15(h) shows that among the mispredictions, most of the points suffer about 10% to 15% loss compared to the best achievable, leading to an average of 99.1% of the best possible runtime on average (GeoMean).

5.6 Chapter Summary

In this chapter, we demonstrate that the design and mapping space of DNN accelerators can be learnt using machine learning models. To be precise, this chapter takes the help of three case studies covering the various example design and mapping space of systolic array based DNN acceleration. In main contributions of the chapter can be summarized as follows. *First*, the work here performs detailed design aware and design agnostic statistical study of the design space to find the underlying patterns in the design space exploration data to make a case for applying a learning model. *Second*, the chapter demonstrates how to formulate the traditional architecture design space exploration problem into a machine learning problem. *Third*, in this chapter we design and train a custom neural network based recommendation engine which is capable of capturing the design and mapping space of all the three case studies presented in this work, and is able to attain high prediction accuracy on test set for all three and greater than the off the shelf ML models.

CHAPTER 6
SELF-ADAPTIVE RECONFIGURABLE ARRAYS (SARA): LEARNING GEMM ACCELERATOR CONFIGURATION SPACE USING ML

6.1 Introduction

Custom architecture design enables us to achieve high performance and energy efficiency for a given class of workloads in post Moore's Law era. Highly specialized architectures however are inflexible to any variation in the nature of workload and thus can easily be rendered obsolete. To mitigate this limitation, there has been an increasing interest in developing flexible architectures which have additional components (interconnects, buffers, and configuration registers) to support changing workload requirements. For popular applications like DNN acceleration, several such flexible architectures have been proposed [8, 11, 68, 20, 9].

In all of the prior works on flexible DNN accelerators, however, the onus of finding and setting the best configuration lies on the software stack, typically using a compiler/mapper [34, 69, 70, 71]. This dependence causes a few deployment challenges: (i) a cost model has be to developed and integrated as an optimizer into the compilation stack to help find optimal mappings, without which the flexible design loses utility, (ii) an expensive configuration and mapping search has to be performed at compile-time before scheduling any workload. Usually mapping search in software is performed via exhaustive, heuristic or optimization algorithm-based approaches which take about a few seconds to hours [69, 34, 71], even with sophisticated ML assisted frameworks like autoTVM [36]. (iii) the search-time overhead also eliminates opportunities for deploying such flexible accelerators for domains/applications with soft or hard-real time inference targets.

In this chapter, we combine the systematic design methodologies developed in Chapter 3

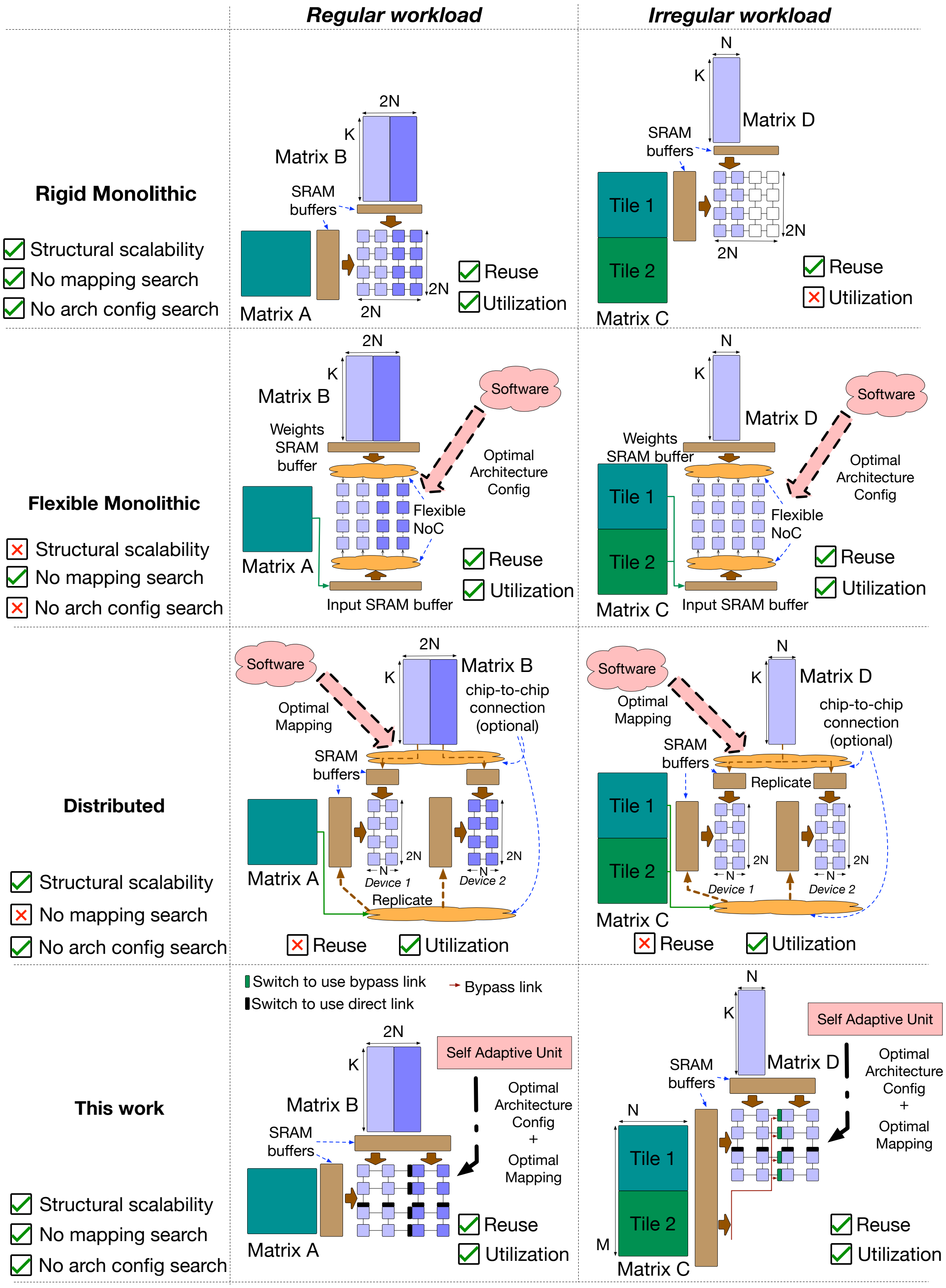


Figure 6.1: Comparison of scalability, utilization, and operand reuse in traditional monolithic and distributed accelerators, and the position of the proposed architecture

and Chapter 4, and the machine learning based design space optimization technique developed in Chapter 5 to create a scalable and yet flexible accelerator which morphs into the best configuration at runtime based on workloads. Specifically, this chapter presents the following two concepts:

First, the chapter describes the work we design a scalable reconfigurable hardware optimized for GEMM workloads called Reconfigurable Systolic Array (*RSA*). *RSA* is developed upon the intuition that flexible accelerators often need to trade-off utilization, data reuse, and hardware complexity (i.e., scalability). This is illustrated in Figure 6.1. *Rigid Monolithic* arrays (e.g., TPU's systolic array [7]), are simple to construct but offer no flexibility leading to high under -utilization for many workloads [17, 59].

Flexible Monolithic arrays (e.g., MAERI [9], Eyeriss_v2 [72], SIGMA [17]) provide flexibility via clever use of interconnects and configuration logic, enabling high utilization for a majority of workloads. However, the increased hardware complexity hinders scaling, and the design requires external software support to exploit the benefits of reconfigurability [69, 70, 34]. Distributed architectures (e.g., Tangram [20], Simba [19]) help address the utilization challenge, since irregular workloads can be tiled on to these smaller arrays. However, this architecture leads to loss in spatial reuse (i.e., direct data-forwarding) that monolithic designs provide, and also requires data replication across the SRAMs of the individual arrays. Data replication leads to a decrease in overall on-chip storage capacity, leading to a loss of temporal reuse due to smaller tiles. Moreover, distributed arrays can exacerbate the mapping search problem [19, 69]. *RSA* aims to address the shortcomings of all three design strategies. It is a flexible accelerator capable of supporting mappings that can be realized by monolithic as well as distributed arrays by configuring to variable array dimensions and number of sub-arrays (as depicted later in Figure 6.5(d)), thereby enhancing both utilization and reuse. In practice, *RSA* closely approximates a flexible monolithic design, with a fraction of area cost.

Second, extending the methods discussed in Chapter 5, we present a custom ML

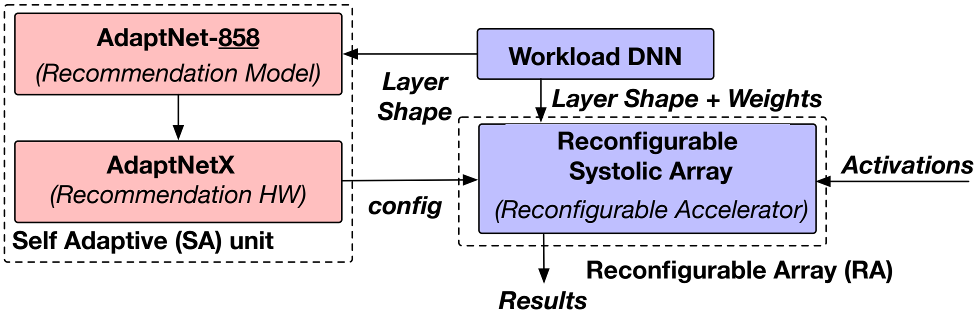


Figure 6.2: The constitution and interactions of the self adaptive (SA) and reconfigurable array (RA) components to make up the SARA accelerator called *SAGAR* in this work.

recommendation system model called AdaptNet that achieves a recommendation accuracy of 95% on a dataset of 200K GEMM workloads, and on average(GeoMean) 99.93% of the best attainable performance (Oracle). We also design a custom hardware unit to run AdaptNet called AdaptNetX. AdaptNetX enables to get a recommendation response for any query in about 600 cycles which is at least about 6 orders of magnitude faster than software. Furthermore, AdaptNetX consumes the same hardware real-estate and roughly the same on-chip memory capacity [1] for different arrays, thus proving to be a scalable solution in contrast to approaches like using configuration caches. With AdaptNetX the configuration lookup using AdaptNet can be performed at runtime, without involving the software stack.

Together, these two components enable us to develop a new class of accelerators that we call *Self Adaptive Reconfigurable Array (SARA)* (Figure 6.2). SARA accelerators can self adapt at runtime to optimized configurations for the target workload, without requiring compile-time analysis. We demonstrate an instance of SARA that we name 'Shape Adaptive GEMM AcceleratoR (*SAGAR* [2]) as shown in Figure 6.2 and evaluate its performance across various configurations.

[1] The only change in AdaptNet between various *RSA* is the weight of the output layer, which is small in comparison to the embedding table which takes most of the on-chip space

[2] means Ocean in Sanskrit, reflecting the shape flexibility of our accelerator.

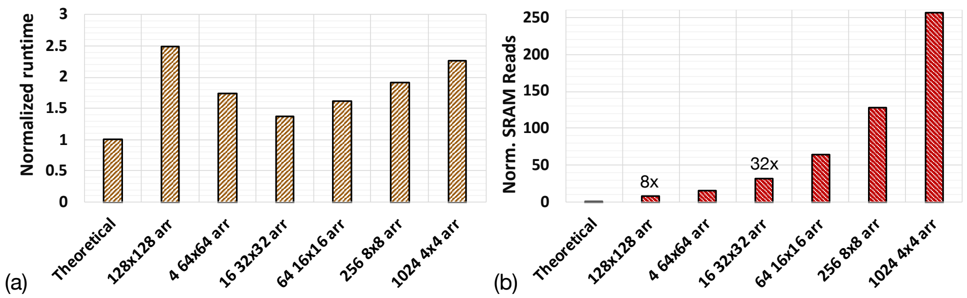


Figure 6.3: The trade-off between improved runtime and lost operand reuse in compute equivalent monolithic and distributed systolic array configurations. (a) the theoretical minimum runtime, and the runtime obtained for stall free operation of monolithic and compute normalized distributed systolic array settings; and (b) the corresponding SRAM reads, normalized to theoretical minimum reads required when multiplying a 256×64 matrix with another 64×256 matrix.

6.2 Background and Motivation

To help understand the trade offs involved in choosing a performant configuration, and the associated loss of reuse we perform a simple experiment. We run one GEMM operation, involving operand matrices of sizes sizes 256×64 and 64×256 on various systolic array configurations. These are, a 128×128 monolithic array, and five distributed scale-out configurations viz. 4 64×64 arrays, 16 32×32 arrays, 64 16×16 arrays, 256 8×8 arrays, and 1024 4×4 arrays. We obtain the runtime and memory accesses for running this workload on all the array configurations using SCALE-Sim [58] (see Section 6.6.1). In Figure 6.3(a) we show the runtime normalized to the theoretical minimum cycles required. Please note that with the chosen matrix dimensions, the systolic arrays in all the configurations are mapped 100% with useful computation. The differences in runtime in various arrays under 100% mapping efficiency is attributed to the array filling and draining at each serialization step (see sec III in [59]). We observe that the configuration with 32×32 array is the most performant, beating the monolithic configuration by about $2\times$. In Figure 6.3(b) we depict the SRAM read accesses performed by all the array configurations, normalized to the theoretical minimum number of reads possible. From this figure we observe that the 32×32 configuration performs about $4\times$ more memory accesses then the monolithic. The excess memory accesses, which lead to reduced energy efficiency, result

Table 6.1: Previous accelerator proposals categorized in terms of computation support, and flexibility of hardware and mapping. Accelerators are categorized into various types introduced in Figure 6.1 viz. Rigid Monolithic (RM), Flexible Monolithic (FM), and Distributed (Dist)

		Mapping Capability		Flexibility			
	Type	Homogenous	Heterogenous	Dataflow	Variable Dimensions	Multi-array Mapping	Self Configurable
Zhang et al. [11]	FM	✓		✓			
Eyeriss [6]	RM	✓					
Alwani et al. [12]	RM		✓				
NeuroCube [73]	Dist		✓			✓	
MAERI [9]	FM	✓	✓		✓		
TPU [7]	RM	✓					
Flexflow [10]	FM	✓		✓			
Tetris [68]	Dist		✓			✓	
Brainwave [8]	Dist		✓			✓	
Simba [19]	Dist		✓			✓	
Tangram [20]	Dist		✓			✓	
Cascades [74]	FM		✓	✓			
Sigma [17]	FM		✓				
Planaria [75]	FM		✓		✓		
***SAGAR* (This work)**		✓	✓	✓	✓	✓	✓

from the loss of wire reuse.

From the discussion above we make two observations.

(i) Distributed arrays are more performant than an equivalent monolithic array, even when mapping efficiency is 100% on both. However, the optimal size of each device in a distributed setting is workload dependent. (ii) Monolithic configurations are strictly more energy efficient than distributed arrays, due to loss the of spatio-temporal reuse in the latter.

It is intuitive to reason that the optimal solution is perhaps a coarser grained array which provides sufficient mapping flexibility while maximizing reuse. In Table 6.1 we inspect a few well known accelerator proposals in terms of scalability and potential to maximize utilization. We notice that simple architectures that are easy to scale in size, under perform on extracting operand reuse. On the other hand, architectures with sufficient flexibility are not scalable. None of the architectures, including the ones with multiple arrays and NoC support, can create variable sized arrays or flexible array dimensions which can help simultaneously achieve high mapping efficiency and data reuse.

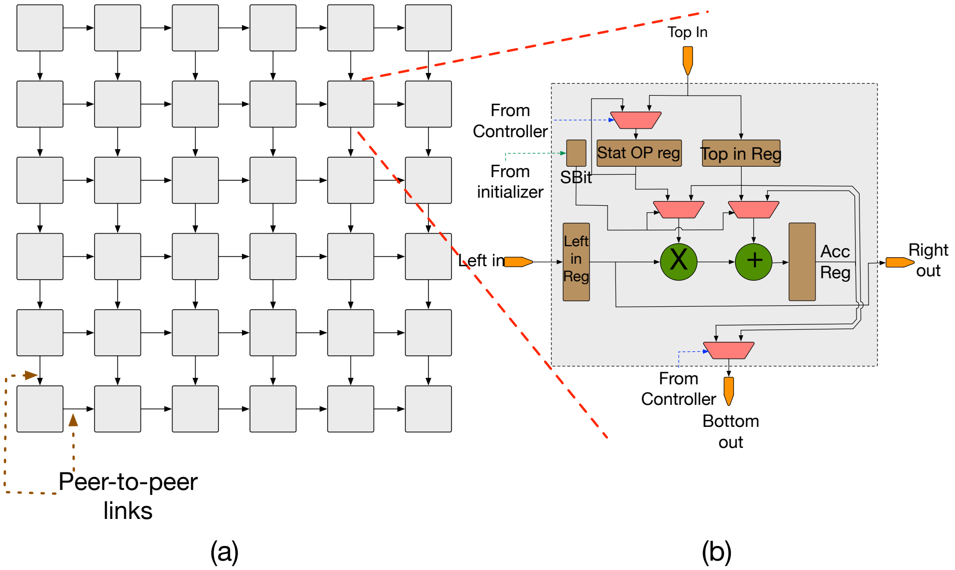


Figure 6.4: (a) A systolic array of traditional MAC units, (b) the architecture of a traditional MAC unit

6.3 Reconfigurable Array Design

6.3.1 Compute array

Traditional MAC units. In Figure 6.4(a) we show a traditional systolic array constructed by laying down Multiply-and-Accumulate (MAC) (Figure 6.4(b)) units in a 2D grid. Each MAC unit is designed to get an operand from either both (*Left in*, *Top in*) ports or from either of the ports, and perform multiplication and addition operation. In the next cycle the operand data received is sent to its neighbour over the peer-to-peer links. The internal registers, and multiplexers enable the array to work in output stationary (OS), weight stationary (WS), and input stationary (IS) modes of operation [6]. This simple mechanism of data movement results in high wire reuse, but at the same time restricts the mapping of compute only to those operations which require same set of operands to be mapped along a row or a column.

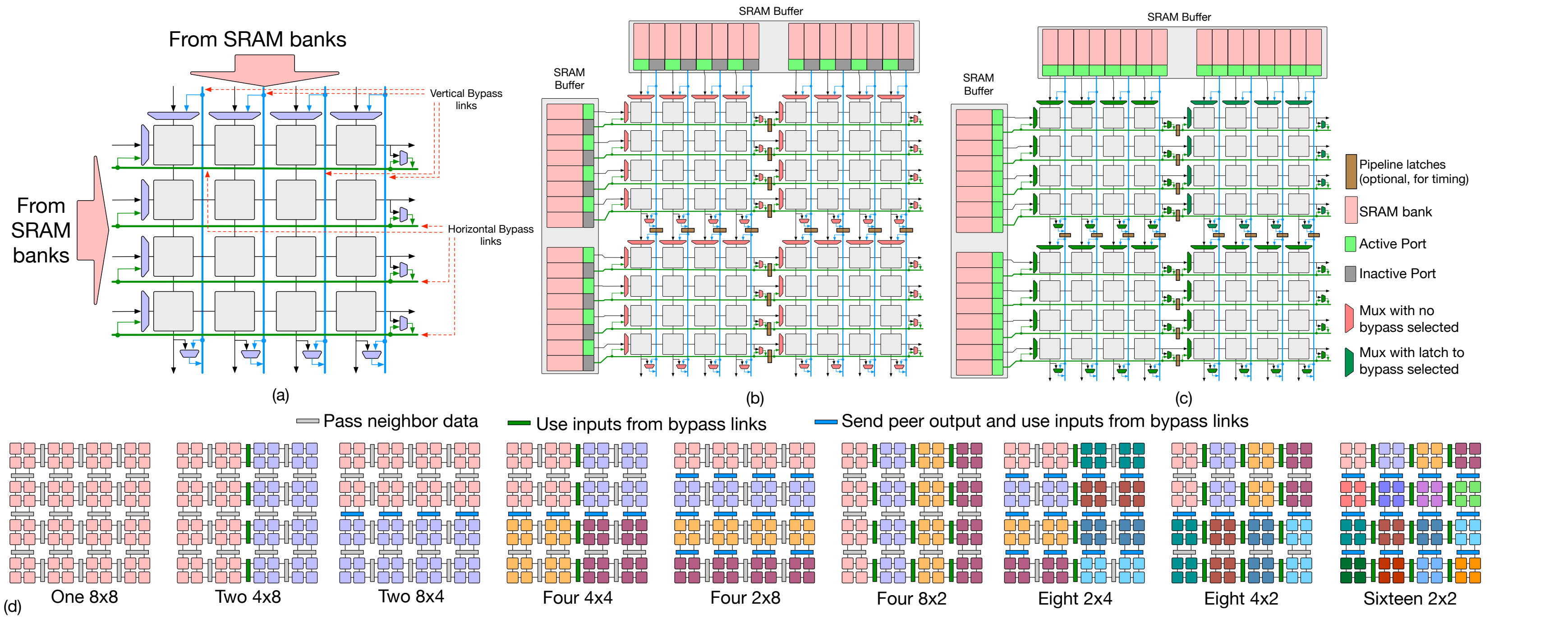


Figure 6.5: (a) Construction of a 4 × 4 *systolic-cell* with bypass muxes and bypass links. (b) A 8 × 8 SMART systolic array operating in scale-up configuration. Each 4 × 4 *systolic-cell* is connected to its neighbor with the peer-to-peer links as the bypass muxes are turned off. The SRAM ports connected to bypass links are unused. (c) Configuration of bypass muxes to enable the 8 × 8 SMART systolic array to work as a scaled-out distributed collection of systolic arrays. The bypass muxes are turned on to allow systolic-cells to directly connect to the SRAM ports which are all active. (d) Possible monolithic and distributed configurations possible in the reconfigurable Smart-Systolic Array(SSA) using 2 × 2 *systolic-cells*

Systolic Cells. The mapping flexibility in systolic arrays can be improved by allowing adjacent MAC units to work on different operands. To enable this, the architecture needs to provision for additional links from the SRAM to the MACs. Providing such links to each MAC however is costly in terms of area as well as energy since the spatial reuse over wires is compromised. To simultaneously achieve mapping flexibility and the advantages of spatial reuse in systolic arrays, we propose a design called *systolic-cell.* A *systolic-cell* is a small grid of traditional MAC units augmented with multiplexers at the edges. This enables them to chose the operands from the neighbouring MAC units or a separate set of operands available via bypass links. The MACs within a *systolic-cell* are connected using peer-to-peer links similar to that of a traditional systolic array. Figure 6.5(a) shows a 4×4 *systolic-cell* example. Please note that choice of the size of a *systolic-cell* is implementation dependent. In general, the smaller the cell size, higher the mapping flexibility, which comes at a cost of slightly increased area and power.

Scale-up and Scale-out using Systolic Cells. Larger arrays can be created by arranging and connecting the *systolic-cell*s as depicted in Figure 6.5(b) using the peer-to-peer links. At the edge of each *systolic-cell* the muxes can be configured to connect to the bypass links. Please note that dedicated bypass links are allocated to each *systolic-cell* to allow concurrency. Attaining flexible mapping in such a design is a matter of configuring the multiplexers of the *systolic-cell*s. Depending on the mapping , an user can chose not to use the bypass paths at all and use the entire array as a single monolithic unit by setting the multiplexers to accept data only to/from the peer-to-peer links, (this is the case depicted in Figure 6.5(b)), which is equivalent to a *scaled-up* configuration. One the other hand, the user can set all the multiplexers to accept and deliver data solely to the bypass links, therefore operating as a cluster of arrays, each the size of a *systolic-cell.* This configuration, depicted in Figure 6.5(c) is equivalent to a *scaled-out* configuration. Figure 6.5(d) illustrates some of the possible configurations constructed using a 64 MAC units with 2×2 *systolic-cell*s. As can be observed in this figure, not only can the array be configured to work in fully monolithic or fully distributed configurations, but also in any of the configurations in between. By setting the appropriate muxes in either pass-through or bypass modes, sub-arrays larger than *systolic-cell* size can be constructed (eg. 4×4, 8×4 etc in this example). Each of the sub-arrays have access to the scratchpad memory using the bypass links. Please note that when fully utilized, a larger systolic array improves energy efficiency over a distributed configuration of same number of MAC units by exploiting wire reuse and reducing SRAM reads. The availability of such variety of choices for reconfiguration leads to flexible and efficient mapping, hence improving the utilization and energy efficiency of the design.

6.3.2 Bypass links

Adding a dedicated bypass link from the SRAM bank to each *systolic-cell* along that row/column is necessary to attain full throughput from the array. Given the nature of the data movements in systolic arrays, we recognize that the vertical links can be used for both second input and the output operands. In Table 6.2, we examine the bandwidth requirements from the bypass links for the three systolic dataflows in a distributed setting, by contrasting it to the requirements of the operands. These requirements clearly dictate that high bandwidth bypass are necessary. Another addition in our proposed architecture are the switches at the edges of the *systolic-cell*s. However, these switches are simple multiplexers, which are configured statically for a given workload, without the need for any additional logic.

Scalability via Pipelining. On-chip wire scalability studies such as SMART[76] have shown that it is possible to traverse a few millimeters (9mm to 11mm) of wire length in 1ns before latching the signal. The authors in SMART achieved this using conventional asynchronous repeaters (a pair of inverters) placed 1mm apart. In *RSA*, repeated wires offer an opportunity to not only cross a single-systolic cell in a cycle, but in fact bypass multiple systolic cells within a single-cycle. In our reference architecture *SAGAR*, we

Table 6.2: Bandwidth requirements for the bypass links for various dataflows, contrasted to the requirements of operands (names in parenthesis reflects the corresponding operands in 2D convolutions)

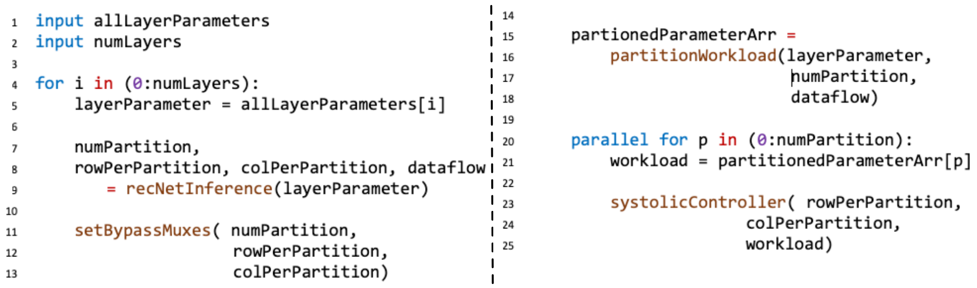
	Operands			Links	
	Input Mat1 (Activations)	Input Mat2 (Filters)	Outputs	Hor. Bypass	Ver. Bypass
Output Stationary	High	High	Low	High (Inputs)	High (Filters)
Weight Stationary	High	Low	High	High (Inputs)	High (Outputs)
Input Stationary	Low	High	High	High (Filters)	High (Outputs)

perform place-and-route to determine the number of systolic cells per pipeline stage of the bypass links. At 28nm, we find that 8 systolic cells can be bypassed at 1GHz, as we demonstrate later in Section 6.6.2, Figure 6.13(h). Note that pipelining the bypass links only adds a few cycles of fill time to the *RSA*, and does not impact the internal timing of the systolic array within each systolic cell (which is itself pipelined at each MAC unit).

6.3.3 Scratch pad memory

The array constructed from *systolic-cells* is backed by SRAM scratchpad memories, which are constructed as two individual buffers. Each of these buffers is dedicated to one of the operand matrices. Such scratchpad SRAM buffers are common in accelerators, and are designed to reduce the number of off chip accesses and facilitate temporal reuse. Each operand buffer is operated in a double buffered fashion, so that the prefetch latency can be minimized. The system also contains a third buffer which is used to store generated outputs elements.

To support the bandwidth of bypass links, we provision for this extra bandwidth by increasing the number of memory banks in the scratchpad SRAM buffers. Despite having the same number of SRAM ports as in a distributed configuration, this approach has a couple of advantages over the latter. *First*, there is no replication of data required, which



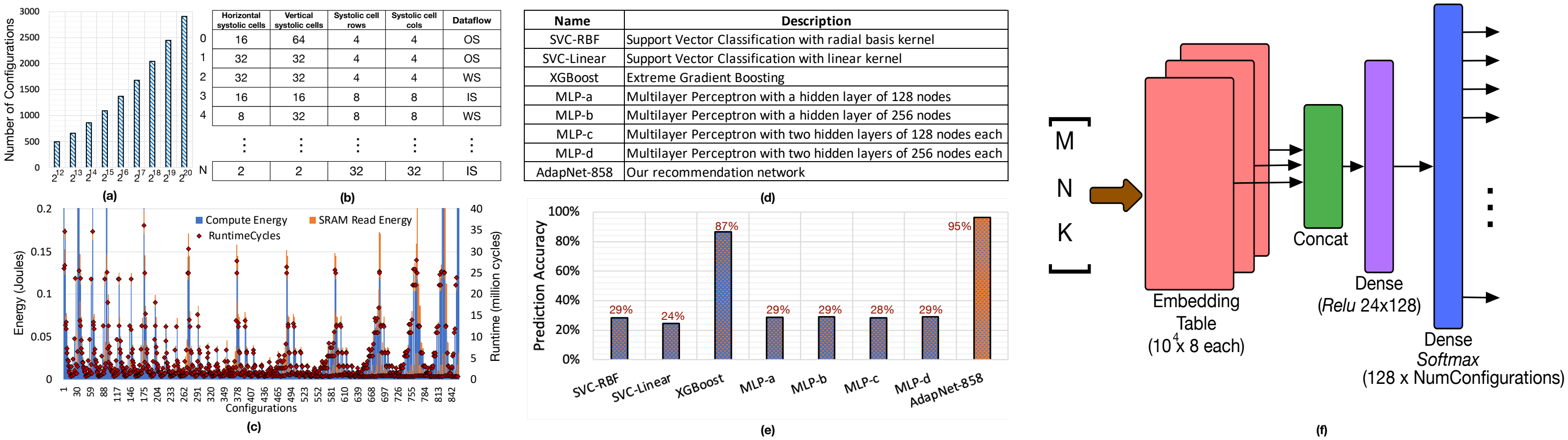
```
input allLayerParameters
input numLayers

for i in (0:numLayers):
    layerParameter = allLayerParameters[i]

    numPartition,
    rowPerPartition, colPerPartition, dataflow
       = recNetInference(layerParameter)

    setBypassMuxes( numPartition,
                    rowPerPartition,
                    colPerPartition)

    partionedParameterArr =
        partitionWorkload(layerParameter,
                          numPartition,
                          dataflow)

    parallel for p in (0:numPartition):
        workload = partitionedParameterArr[p]

        systolicController( rowPerPartition,
                            colPerPartition,
                            workload)
```

Figure 6.6: Psuedocode depicting the control logic

otherwise reduces the effective capacity of the system therefore adversely affecting reuse. In our design by eliminating replication we inherently improve the temporal reuse of operands. *Second*, each of the *systolic-cells* can access data in the entire operand buffer. Due to unified memory control of each buffer, operation like multi-cast are implicit in form of read collation, which improves energy efficiency without impacting performance. We describe the impact on reads and energy efficiency in detail in Section 6.6.1.

6.3.4 Control

Figure 6.6 shows the control logic executed for each GEMM workload or DNN layer. The control logic of our proposed system is similar to the control of a distributed systolic array based system. However, unlike other systems, in a *systolic-cell* based design, the number of distributed units is a variable and is determined at runtime based on the data-flow and operand shapes. The following steps describe the logic. **1.** recNetInference(): In this work we use a recommendation system based described in Section 6.4. The model takes in the layer parameters and recommends a configuration, which is the most efficient for the workload. **2.** setBypassMuxes(): Next, the bypass muxes are set in the compute hardware to realize the partitioned configuration. This is accomplished by writing select values to a register, whose individual bits drives the select lines. These configurations stay static throughout the GEMM computation. **3.** partitionWorkload(): The control logic, then partitions the original workload by marking portions of the original operand arrays to be used by each individual partition. **4.** systolicController(): Finally, for each partition, an instance of systolic array controller is initiated to drive the GEMM operations to completion and orchestrate the required data movement. Please note that in contrast to a traditional systolic array like TPU, multiple control units are required to work in parallel.



	Horizontal systolic cells	Vertical systolic cells	Systolic cell rows	Systolic cell cols	Dataflow
0	16	64	4	4	OS
1	32	32	4	4	OS
2	32	32	4	4	WS
3	16	16	8	8	IS
4	8	32	8	8	WS
	⋮	⋮	⋮	⋮	⋮
N	2	2	32	32	IS

Name	Description
SVC-RBF	Support Vector Classification with radial basis kernel
SVC-Linear	Support Vector Classification with linear kernel
XGBoost	Extreme Gradient Boosting
MLP-a	Multilayer Perceptron with a hidden layer of 128 nodes
MLP-b	Multilayer Perceptron with a hidden layer of 256 nodes
MLP-c	Multilayer Perceptron with two hidden layers of 128 nodes each
MLP-d	Multilayer Perceptron with two hidden layers of 256 nodes each
AdapNet-858	Our recommendation network

Figure 6.7: (a) Size of the configuration space wrt number of MAC units for a *systolic-cell* based flexible array (b) Example of configurations predicted by ADAPTNET indexed by category ID (c) Performance and energy consumption by various configurations when running layer 19 of FasterRCNN (d) Chart showing the name and description and name of the various classifiers used in this work (e) The accuracies obtained by classifiers on predicting the architecture parameters for out dataset of RSA configurations with 2^{14} MAC units (f) Architecture of our proposed recommendation network

6.4 Recommendation Model

This section describes a neural network based recommendation unit which can simultaneously predict the optimal architecture configuration and mapping strategy, when a workload arrives. This system solves two problems. First it minimizes the changes required in a compiler for configuration and mapping search, thus easing deployment. Second, it enables real time reconfigurability. Given that a large reconfigurable array is most likely be deployed in data-center like use cases, the capability to adapt in real time will help achieve improved resource allocation and consequently meeting tight service-level-agreements (SLA).

6.4.1 Architecture design as ML problem

To facilitate learning the design space we have to frame the search problem into a ML task framework like classification or regression. We found that framing this as a classification or recommendation task works the best. This abstraction lets us leverage the existing works and models which have been invented by the ML community. An important step in solving this problem is to define the output space of the model. It is natural to assign bins for the each of the design parameters and independently predict the optimal values for each parameter of interest. However, this would require a separate model to be trained and queried for each design parameter. We show that multiple parameters can be combined into a single output class and consequently can capture the design space using a single model.

In our case, the output space comprises of (i) The number and logical layout of the partitions, (ii) The dimensions of the arrays in each partition, and (iii) the mapping/dataflow to be used eg. output stationary (OS), weight stationary (WS),and input stationary (IS). Figure 6.7(b) shows this output space captured as categories of architecture configurations, indexed by the class ID. The learned classifier is expected to select an architecture configura-

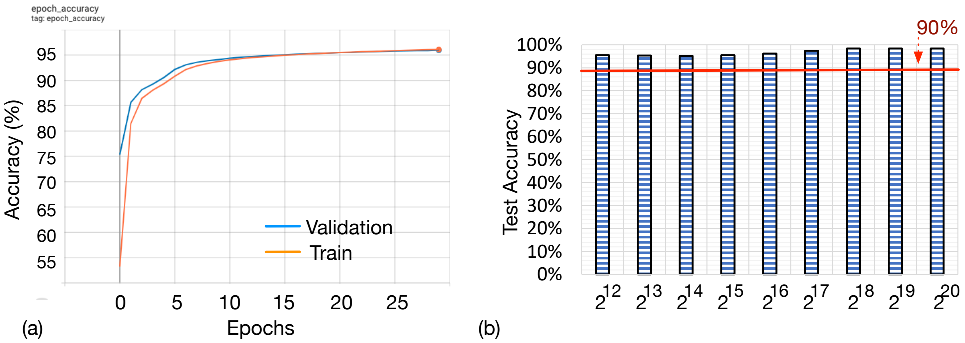


Figure 6.8: (a) The training and validation accuracies obtained during the training step in AdaptNet (b) Test accuracies obtained on test sets for AdaptNets trained on *systolic-cell* based flexible array with various MAC units

tion and corresponding mapping strategy which provides the optimal performance for the workload. To better visualize the complexity of the design space, in Figure 6.7(c) we depict the runtime and energy consumption for computation and SRAM reads when running layer 19 of FasterRCNN (see Section 6.6.1) for the different architecture configurations and dataflows. We observe that determining the optima is a non-trivial task; and the likelihood to chose a sub-optimal configuration is high, when naive methods are used, resulting in significant performance and energy costs. Moreover, the best configuration for this layer is using 256 partitions laid out as a 8×32 grid of 16×4 arrays using WS dataflow, which does not conform to conventional practice of using square or near-square layouts.

6.4.2 Recommendation Neural Network

Dataset generation. We generated a dataset of about 2 million workloads, by sampling M, N, and K dimensions from a uniform distribution of positive integers $<= 10^4$. For each such workload dimension we searched through the configuration space of the reconfigurable array design using 2^{12} MAC units to find the optimal (minimum runtime) configuration using SCALE-Sim simulator, modified for fast runtime estimation. When using a server cluster with about 200 Xeon cores, it takes about a week to obtain all the samples.

Choosing the classifier. Abstracting the problem in the form of a classification naturally opens up the choice of using existing classification algorithms. We explored a handful of pre-existing classifiers, some of which are listed in Figure 6.7(d). The Support Vector Classifiers and the XGBoost models we use are standard implementations provided in scikit-learn [60] and xgboost[62] python packages respectively. We implement the MLPs in keras subpackage in tensorflow and train them for 20 epochs. In Figure 6.7(e), we show the prediction accuracy of these models on a test set of 200K points, after the model has been trained on 90% of the dataset. It is interesting to observe that among all the models only XGBoost was able to reasonably learn the design space and achieve about 87% prediction accuracy.

Recommendation Model: The performance of XGBoost model is encouraging and demonstrates that the design space can be learnt. To further improve the prediction performance of the model, we hand designed a recommendation neural network. We take inspiration from typical neural network based recommendation systems like DLRM[66], which is constructed by augmenting embedding lookups with MLP based classification. The presence of trainable embeddings help in mapping the input data from the raw input space to a latent space, which is observed to improve the classification performance. Given our use-case, there are two main requirements we need to satisfy. First, we need our network to have high accuracy in predicting the best runtime configuration which maximizes performance. Second, given that the recommendation network needs to be queried at runtime, the network should be small keep the inference latency and implementation costs low. In our use case, the recommendation inference for a given layer is run concurrent to the execution of a previous layer whenever possible. Lower inference latency therefore moves the recommendation step out of the critical path. Moreover, a smaller network has fewer computation and storage requirements and hence minimizes the overheads. Honoring these requirements, we propose a network as depicted in Figure 6.7(f). The network, called AdaptNet, is simple, where we lookup the embedding entries for the input features, and then use a classifier with single hidden layer with 128 nodes and softmax activation at the output.

Training, Performance, and Generalizability. To train our recommendation network we use one Titan RTX GPU with 84 SMs. When training on the dataset for 2^{14} MAC based RSA, for 30 epochs with a mini-batch size of 32, it takes about an hour to converge. Figure 6.8(a) shows the accuracy progression as the training proceeds. We obtain a high accuracy of 95% of the test dataset of 200K points, which is compared against other classifiers in Figure 6.7(e). We also test the robustness of our design by generating similar datasets of 2M points each for RSA's with varying number of MAC units (eg 2^{12}, 2^{13} etc). The aim is to test the performance of different AdaptNet with different output configuration space. In Figure 6.8(b) we plot the test accuracies obtained for each such AdaptNet trained for 30 epochs with 90:10 training-testing split. Please note that the data points in test datasets are unknown at training time. We observe that the networks all achieve high accuracies over 90%. To distinguish the AdaptNet's among themselves we use the size of the configuration space as a suffix. For example, the design space of 2^{14} MAC has 858 possible configuration, therefore we call the corresponding network AdaptNet-858.

6.4.3 Alternatives to AdaptNet

Memoization, in form of caching is one alternative to AdaptNet to attain constant time configuration lookup. However, caching only works for a limited number of previously computed workloads. For any workload which does not hit in the cache, search has to be performed at runtime. The large configuration space of *RSA* as depicted in Figure 6.7(a) makes it a non scalable solution. One the other hand, AdaptNet, owing to learned parameters, can generalize configuration recommendation to any query having workload dimensions generated from the distribution of its training dataset.

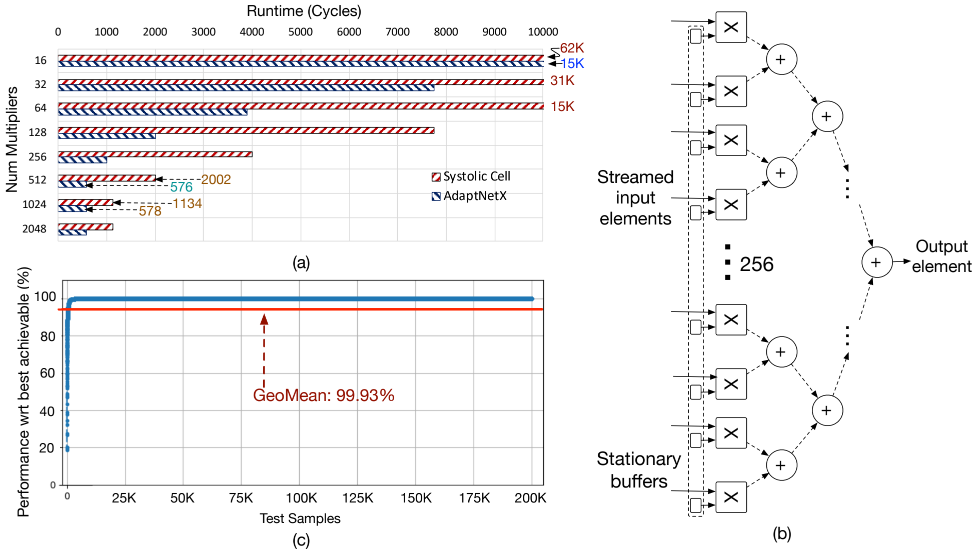


Figure 6.9: (a) Cycles needed to run AdaptNet-858 on an array of *systolic-cell*s and on the custom hardware unit (AdaptNetX) as a function of number of multipliers. (b) Architecture of the custom 1-D unit hardware for AdaptNetX(c) Relative performance of the configurations predicted by AdaptNet-858 for *SAGAR* for 2×10^5 test samples when compared to the runtime of best possible configurations

6.5 Self Adaptive Reconfigurable Arrays

By coupling AdaptNet with a reconfigurable array, we can create a self adaptive system which can be conceptually viewed as a combination of two units, a Self Adaptive unit (SA), and a Reconfigurable Array (RA) unit as shown in Figure 6.2. The SA unit encompasses the software and hardware components which recommend the optimal configurations. The RA unit is the hardware unit capable of flexibly configuring to the recommended configurations and hence run the workloads. It is worth pointing out that this design class is not specific to a reconfigurable core for running GEMM workloads. Instead any Coarse Grained Reconfigurable Array (CGRA) unit, configurable at runtime, can be augmented with a suitable SA, to ensure optimal performance. We believe this results in a new class of designs, which we name Self Adaptive Reconfigurable Array (SARA).

6.5.1 Hardware to run AdaptNet

In the context of our use case, an intuitive option is to allocate a few *systolic-cell*s from the main array to run AdaptNet. However, this choice will lead to either fewer MAC units left for the actual workloads, or to allocate additional *systolic-cell*s for AdaptNet leading to an additional overhead. An alternative to adding more *systolic-cell*s will be to add a custom hardware dedicated for running AdaptNet. We explore both the *systolic-cell* and custom hardware options below for AdaptNet-858.

AdaptNet Runtime on *systolic-cells*. Figure 6.9(a) shows the cycles required for a single inference of the AdaptNet as a function of multipliers used in 4×4 *systolic-cell* based array. Understandably, the runtime decreases proportional to the increase in number of multipliers as we increase the number of *systolic-cell*s, achieving the best runtime of 1134 cycles when using 1024 multipliers or 64 cells. When both the workloads and the recommendation engine is run on a same array; for a TPU equivalent machine with 2^{14} MAC units, about 6.25% of the array needs to be allocated for running the AdaptNet. Another choice could be allocating more hardware resources in terms of extra 64 *systolic-cell*s dedicated to run the recommender network. However, given that AdaptNet has exclusively dense layers processing the embedding lookups, a systolic execution turns out to be sub-optimal.

AdaptNet Runtime on AdaptNetX. We found a custom design tuned for AdaptNet layer parameters to be more efficient. For efficient execution of the dense layers, we chose a 1-D multiplier unit with a binary tree based reduction as shown in Figure 6.9(b). We found Input stationary (IS) dataflow to be the most performant for our use case. In this mapping the elements of the input vector is buffered near the multipliers, while elements of the weight matrix are streamed through to generate one output element/partial sum, with a sustained throughput of 1 element per cycle. Throughput can be further increased by adding more such 1-D units. We name the custom core with one or more such 1-D units as AdaptNetX. In Figure 6.9(a) we depict the variation of runtime of AdaptNet inference on AdaptNetX with two 1-D units as a function of multipliers. We find the 512 multipliers result in best runtime of 576 cycles, when running AdaptNet for 2^{14} MAC unit *systolic-cell* design. We also examine the cost of misprediction of AdaptNet in Figure 6.9(c), where we plot the runtime of the predicted configurations from AdaptNet-858 normalized to best possible runtime. We see that most mispredictions are benign and only a few misprediction lead to catastrophic performance losses, leading to a geometric mean of 99.93% of the best possible performance.

6.5.2 *SAGAR* Accelerator

SAGAR is constructed by augmenting the 2^{14} MAC *RSA* unit, laid out as 32×32 grid of *systolic-cell*s, with AdaptNetX running AdaptNet-858 (see Figure 6.10). We chose this configuration as it has the same compute as the TPU v2, and the 4×4 *systolic-cell* size works the best for our workloads (see Section 6.6.1). Since each row and column in this configuration has 31 bypass links and one link to MAC, each buffer is constructed as a collection of 1024 1KB banks.

Real-time Reconfiguration. The AdaptNetX uses an additional SRAM bank of 512KB to store the embedding table and the weight matrices for AdaptNet-858. Each configuration corresponds to a 3968 bit vector which sets the bypass muxes, once the layer is ready to be mapped.

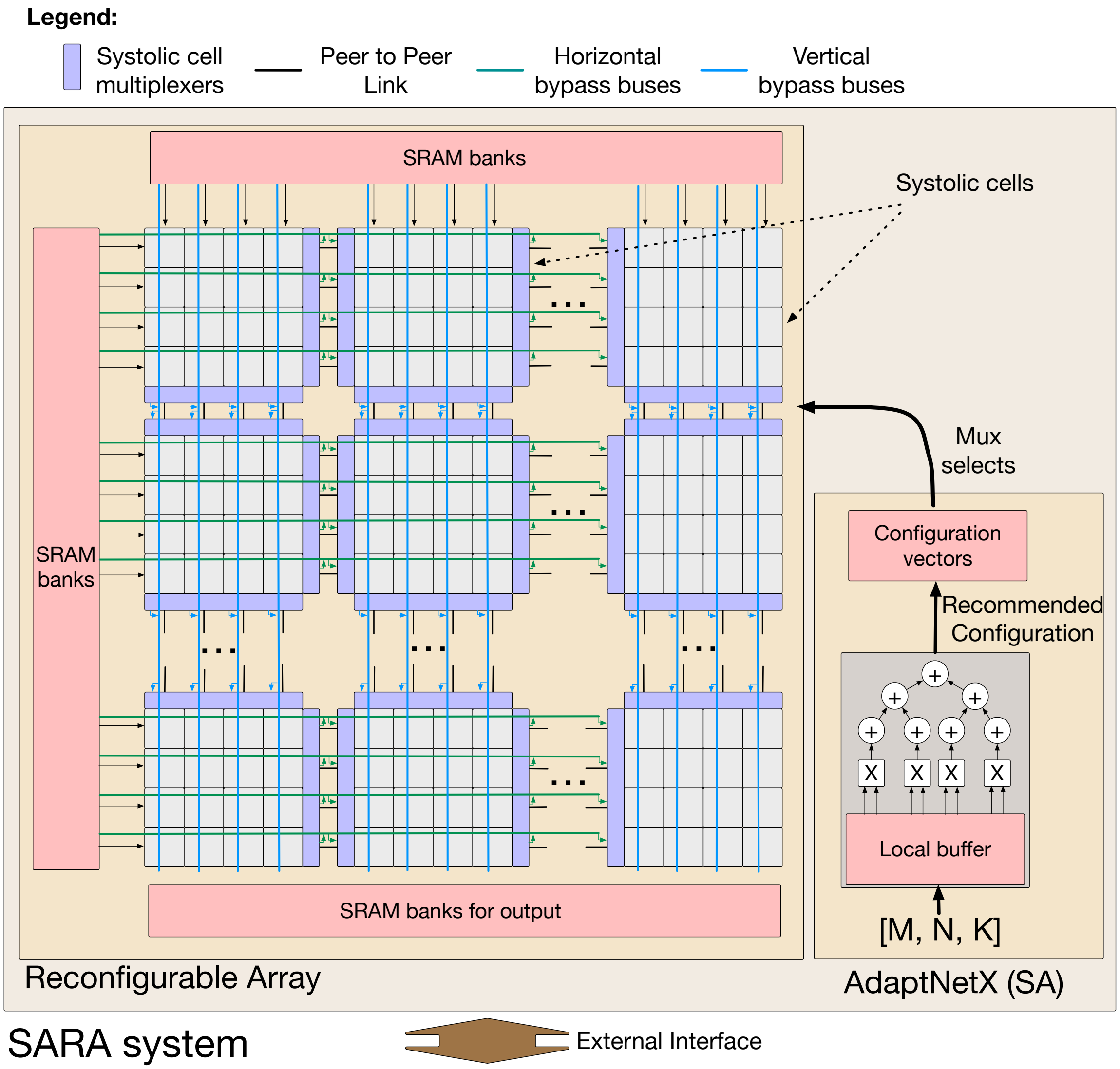


Figure 6.10: Schematic of *SAGAR*, an instance of a SARA accelerator.

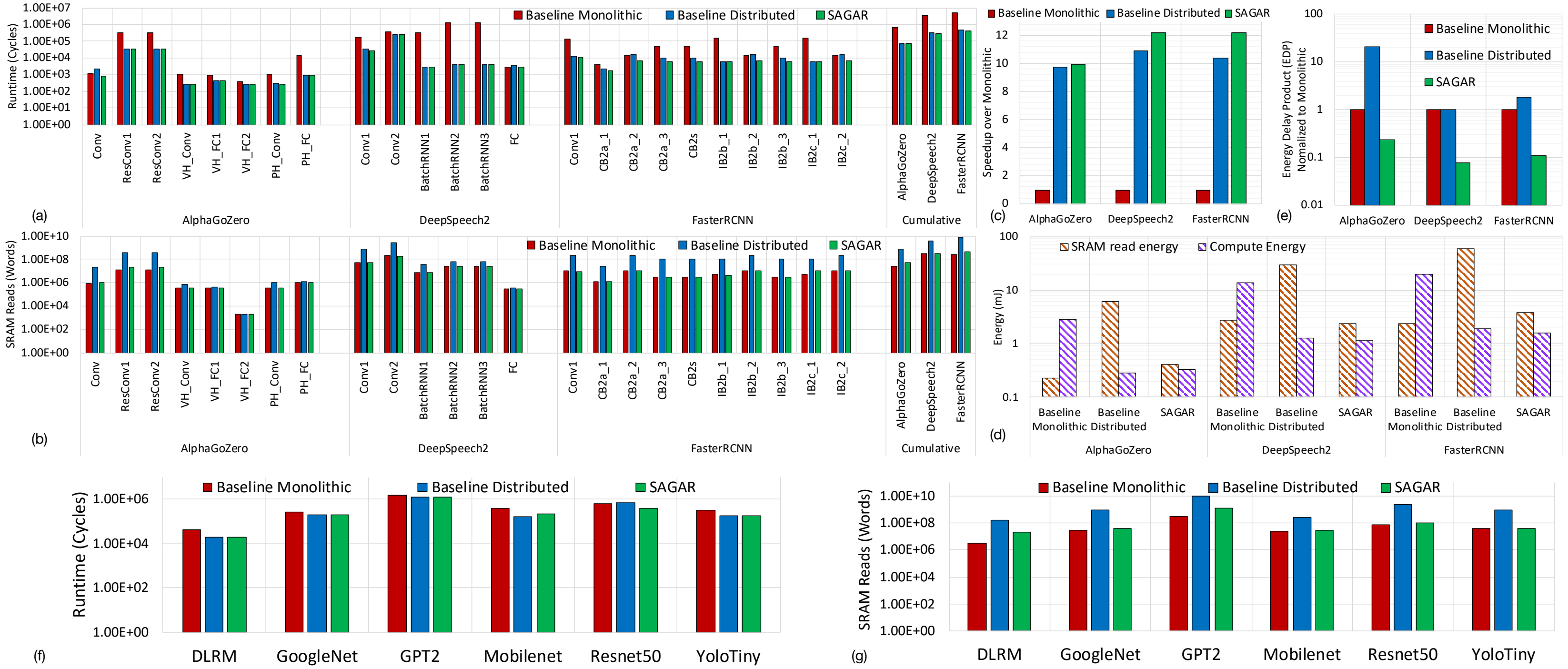


Figure 6.11: (a) Simulated runtimes for monolithic 128 × 128 baseline, distributed 1024 4 × 4 baseline, and *SAGAR* for layers in AlphaGoZero, DeepSpeech2, and first 10 layers of FasterRCNN (b) SRAM reads for the same workloads for *SAGAR* and baseline configurations (c) Speedup of *SAGAR* and distributed baseline as compared to the monolithic baseline (d) Energy consumption breakdown for our workloads in *SAGAR* and baselines (e) Energy delay product(EDP) of *SAGAR* and baselines, normalized to EDP for monolithic baseline (f-g) Sensitivity analysis on various networks for runtime and SRAM reads

Table 6.3: Table depicting the architectural configuration of distributed systolic array based systems, monolithic systolic array baseline, and *SAGAR*

Name	Num Units	MAC/unit	Banks per SRAM buffer	Capacity per SRAM bank
Dist. 4x4 units (Baseline)	1024	16	4	256 B
Dist. 8x8 units	256	64	8	512 B
Dist. 16x16 units	64	256	16	1 KB
Dist. 32x32 units	16	1024	32	2 KB
Dist. 64x64 units	4	4096	64	4 KB
Monolithic 128x128 (Baseline)	1	16384	128	8 KB
SAGAR	**1**	**16384**	**1024**	**1 KB**

6.6 Evaluations

We evaluate *SAGAR* in two settings. To capture the merits of the architecture, we present results obtained from simulation. While the implementation aspects are captured by reporting PPA number obtained from Place-and-Route (PnR).

6.6.1 Architectural evaluations

Methodology. For our architecture level studies we chose to use SCALE-Sim [58]. SCALE-Sim is a cycle accurate simulator for systolic array, which generates per cycle data accesses to and from various memories. This enables us to estimate and compare performance, energy consumption, power etc. of systolic array based components to a certain degree of accuracy. We created in-house scripts to generate SCALE-sim input files to perform the workload partitioning for the configurations recommended by AdaptNet-858.

Workloads. For our evaluations we choose FasterRCNN[67], DeepSpeech2[48], and AlphaGoZero[77], as our workloads as a representative of convolution neural networks, language modelling network, and DNNs for reinforcement learning respectively. Figure 6.11(f-g) shows our sensitivity analysis using a few other well known networks.

Table 6.4: Dimensions for the synthetic GEMM workloads

	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12	G13	G14	G15	G16	G17	G18	G19	G20
M	128	256	512	1024	2048	128	256	512	1024	2048	64	64	64	64	64	64	64	64	64	64
K	128	256	512	1024	2048	64	64	64	64	64	64	64	64	64	64	128	256	512	1024	2048
N	128	256	512	1024	2048	64	64	64	64	64	128	256	512	1024	2048	64	64	64	64	64

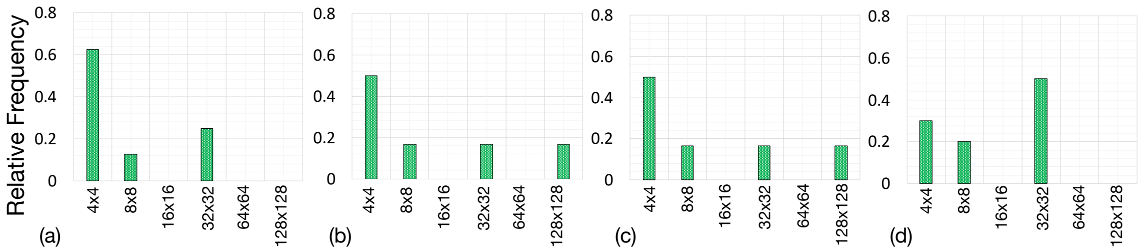


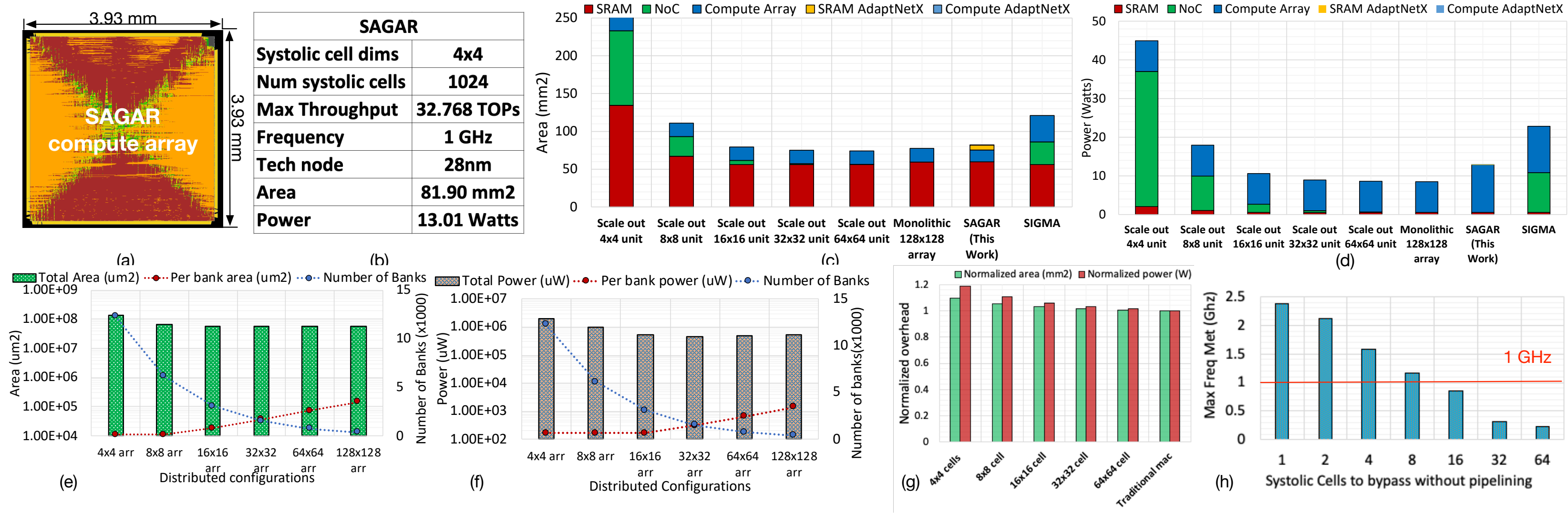
Figure 6.12: Distribution of favorable array sizes for a 16384 MAC distributed system which attain the lowest runtime when run for each layer in (a) synthetic GEMM workloads (b) AlphaGoZero, (c) DeepSpeech2, and (d) FasterRCNN.

Baselines. We chose a 128 × 128 monolithic systolic array and distributed array of 1024 4 × 4 arrays as our baselines as depicted in Table 6.3. Both the arrays have same number of MAC units as TPUv2. Each array in distributed configuration resembles the tensor cores in Nvidia GPUs. Both that baselines have the same total SRAM memory capacity of 3MB divided into buffers for staging two operand and one output matrix.

Performance Analysis. We model both of the baseline systems and *SAGAR* in SCALE-Sim and compare the performance for our workloads. In Figure 6.11(a) we depict the cycles taken to run all the layers in AlphaGoZero, DeepSpeech2, and the first 10 layers of FasterRCNN networks. Among the baselines, the distributed configuration mostly results in faster runtime owing to higher mapping flexibility. However *SAGAR*, owing to reconfigurability is capable of matching the better baseline configuration. Naturally, this flexibility leads to lower aggregated runtime for *SAGAR* than either of the baselines. We see this trend generalizing in Figure 6.11(f) as well.

***systolic-cell* Design Space Exploration.** *SAGAR* is also capable of realizing configurations which are out of scope of either of baselines. This allows *SAGAR* to achieve higher performance than both the baselines on certain layers. For example, consider the synthetic GEMM operands depicted in Table 6.4. Figure 6.12(a) depicts the histogram of the best configuration for these layers obtained from simulation. The layers favouring 8×8 or 32×32 configurations constitute about 40% of the set. Neither of these configurations can be realized a fixed array configuration like the baselines. In Figure 6.12(b,c,d) we show the histogram of a similar experiment conducted on our DNN workloads. For these specific workloads, the 4×4 configuration works the best for majority of the layers. This observation also explains our findings in Figure 6.11(a) on why *SAGAR*'s performance is identical to the 4×4 baseline. Nevertheless, for layers which favor configurations like 8×8, 32×32 etc. *SAGAR* will lead to lower runtime than both the baselines. This is depicted by Figure 6.11(c), where we see that *SAGAR* achieves about $> 10\times$ speedup over monolithic when distributed configurations are preferred. While in cases where monolithic is preferred it runs faster than both the baselines.

SRAM reads and Energy efficiency. In general, due to the loss of reuse, distributed configurations with smaller array sizes have more SRAM reads resulting in lower energy efficiency. We observe this trend in action in Figure 6.11(b) where we depict the number of SRAM reads performed for layers when running our workloads on the two baselines and on *SAGAR*. The distributed 4×4 system has much higher number of reads as compared to *SAGAR* and the monolithic baseline. In *SAGAR* this efficiency loss in reuse is mitigated by using bypassing links. As shown in Figure 6.11(b), across all layers in our workloads, *SAGAR* incurs SRAM reads close to that in the monolithic baseline. In the case of DeepSpeech2, *SAGAR*, owing to efficient mapping, incurs reads even fewer than that of the monolithic baseline. Similar trends are also reflected in other networks as well (Figure 6.11(g)). To further quantify the efficiency of *SAGAR*, we estimated the energy spent by the three configurations on the workloads by taking into account the cycle counts and the SRAM reads and scaling the counts by typical energy consumed per operation computed from RTL PnR flows. For all the workloads, the wire energies calculated using 100 fJ/bit-mm at 14nm [78], come to be about 0.1% (maximum being 0.11% or 0.8uJ in AlphaGoZero), which is negligible. In Figure 6.11(d) we plot the energy consumed for the three workloads on the baselines and *SAGAR*. We observe that for workloads amenable to monolithic array (ie. FasterRCNN and DeepSpeech2), *SAGAR*'s energy consumption is almost identical to the monolithic baseline. The distributed baseline on the other hand consumes an order of magnitude higher energy for all the three workloads, while supporting the same mapping configurations as *SAGAR*. The difference in energies are a direct consequence of utilization. Since fine grained power or clock gating is impractical, the arrays with poor utilization consume same amount of power as the arrays with better utilization. However, these arrays take longer to complete resulting in higher energy consumption. For AlphaGoZero, which favours a distributed configuration, *SAGAR* consumes about 20% of the energy consumed by the monolithic baseline, while almost one order of magnitude lower than that of the distributed baseline. Figure 6.11(d) also shows that *SAGAR*'s energy consumption for SRAM is close to that of consumed by the monolithic array for all the three workloads. The computation energy consumption in *SAGAR* equivalent to the better of the two baselines. The combined effect of improved latency and reuse is perhaps better represented by the energy-delay product (EDP) depicted by Figure 6.11(e). In this figure we plot the EDP for *SAGAR* and the two baselines normalized to the values corresponding to the monolithic configuration. We observe that *SAGAR* results in about 92% to 80% less EDP compared to the monolithic baseline. This further demonstrates the efficiency of our proposed architecture, resulting from preserving reuse while simultaneously decreasing latency due to improved mapping.



SAGAR	
Systolic cell dims	4x4
Num systolic cells	1024
Max Throughput	32.768 TOPs
Frequency	1 GHz
Tech node	28nm
Area	81.90 mm2
Power	13.01 Watts

Figure 6.13: Design-space exploration and final architecture of *SAGAR*. (a) The post PnR floor-plan diagram of *SAGAR*'s compute array, (b) A table detailing architecture configuration of *SAGAR*, the implementation parameters, and post PnR area and power of *SAGAR*. (c) The comparison and breakdown of post synthesis area for distributed systolic array based designs, the monolithic systolic baseline, *SAGAR*, and SIGMA (d) The corresponding breakdown for power consumed by various components in distributed systolic array based designs, the monolithic systolic baseline, *SAGAR*, and SIGMA (e) The variation of total area footprint of SRAM banks in various distributed systolic array and monolithic configuration juxtaposed with the variation in bank sizes and the number of banks required, (f) A similar variation in the power consumption by the SRAM banks in distributed systolic array and monolithic configurations, and (g) the the area and power of a 128 × 128 array when constructed using different sized of "*systolic-cells*" normalized to the area and power of an array constructed with traditional MAC units. (h) The maximum attainable frequency vs the number of 4 × 4 *systolic-cell*s to bypass at 28nm.

6.6.2 Implementation evaluations

Methodology. We implemented *SAGAR* in RTL as a 32×32 array of 4×4 *systolic-cells* and ran ASIC flow till Place-and-Route (PnR) to obtain area and power. We used 28nm library for implementing the logic. We also implemented the SRAM buffers as a collection of 1024 1KB cells with the SAED32 education library from Synopsis, to quantify the power and area overheads, and then scaled down to 28nm equivalent by using Dennard's scaling [79]. Figure 6.13(a) depicts the post PnR floorplan of *SAGAR*'s compute logic. Figure 6.13(b) lists the array configuration, area, and power consumption reported after PnR by synthesizing the *RSA* and memory at a operating frequency of 1 GHz. At 32.768 TOPs (with 1 MAC being two operation) at 1 GHz *SAGAR* takes 81.90 mm^2 of real estate while consuming 13.01 W of power. AdaptNetX consumes 8.65% of area and 1.36% of power.

Baselines. We implement the baseline monolithic 128×128 systolic array and distributed 4×4 array in RTL. The distributed array is implemented using 1024 identical 4×4 traditional systolic arrays connected together by a mesh interconnect. We used the OpenSMART [80] tool to generate and synthesize the mesh topologies for these systems.

The total memory capacity of both the monolithic and the distributed configurations are kept the same at 3MB. As discussed in Section 6.6.1 the monolithic array has two input operand buffer of 1MB each and an output buffer also with the 1MB capacity. In our implementation, we opted for one bank per row or column of the array. This choice ensures that each incoming link to the array will have full bandwidth from SRAM provided that bank conflicts are negligible. Therefore each buffer in the monolithic baseline is constructed using 128, 8KB banks. For the distributed configuration, for each 4×4 array we end up with 1MB for each operand buffer. Using the same design approach as above, we end up with each buffer being constructed using 4 banks of 256 words each. In Table 6.3 we extend the same design principle for designing the memory for various other cell sizes and for *SAGAR*. In *SAGAR*, in addition to the links going directly from the SRAM to the edge MAC units of the array, we have to consider the bypass links as well. To get full bandwidth on these links we need to consider additional buffers. Extending the design described in Figure 6.5, each row and column of *SAGAR* has 31 bypass links and one link to the first MAC unit, we need 32 banks per row/column. Therefore each SRAM buffer is constructed with 1024, 1KB banks.

Area Analysis. In Figure 6.13(c) we depict the break down of area overheads for SRAM buffers, mesh NoC, and the compute array for various distributed configurations, the monolithic array, *SAGAR* and SIGMA [17]. We observe that the monolithic configuration is the most efficient in terms of area, where it is about 5× more compact than the distributed 4×4 array configuration. The breakdown suggests that the bloating in the distributed 4×4 configuration is caused predominantly by the Mesh NoC (contributing to 40.5%), followed by the SRAM buffers. *SAGAR* on the other hand takes about 8% more area than the monolithic array, while consuming about 3.2× lower area than the distributed 4×4 configuration. Considering both *SAGAR* and the distributed configuration provides same mapping flexibility, the proposed design is strictly more efficient.

Across the various systolic-array configurations in Figure 6.13(c), the SRAM area appears to remain fairly constant. This is a direct consequence of the buffer capacity and construction of the array. In Figure 6.13(e) we depict the total area obtained for various configurations depicted in Table 6.3. We observe that, the various configurations vary in the bank capacity and the number of banks. Since the total capacity remains the same across the configurations, these factor counter balance each other leading to observed trends.

Power Consumption. In Figure 6.13(d) we depict the post PnR power consumption for various array configuration. The Mesh NoC stands out as the major contributor, which naturally makes the 4×4 distributed configuration about 5.3× more expensive than the monolithic configuration, with the NoC contributing to about 78% of the power. Considering the power of the compute array alone, all the systolic-array based configurations appear to consume similar power. We also depict the trend in power consumed by SRAM banks across various systolic-array based configurations in Figure 6.13(f). Similar to the trends observed in area breakdown, the counter balancing affects of increasing the bank sizes and lowering of number of banks lead to similar powers across various distributed and monolithic configurations.

RSA however consumes about 50% more power than the monolithic configuration, owing to the bypass links. However this extra cost results in achieving the same mapping flexibility of the 4×4 distributed configuration, which is about 3.5× more expensive.

Scalability Analysis. (i) Figure 6.13(g) we show the overhead of using smaller *systolic-cell* sizes in terms of area and power normalized to monolithic configuration. For specific use cases with relaxed requirements for flexibility larger sized *systolic-cell*s can be used to improve the implementation costs. (ii) Figure 6.13(h) we depict the max frequency that can be met as a function of number of 4×4 *systolic-cell*s that can be bypassed at 28 nm. Since we target 1GHz, we need to pipeline the bypass paths by inserting flops after 8 *systolic-cell*s as we discuss in Section 6.3.2.

6.6.3 Comparison with SIGMA

Implementation Comparison. We compare the area of *SAGAR* with the published area and power numbers of a state-of-the-art flexible accelerator SIGMA [17]. SIGMA allocates a significant portion of area for NoC, which together with SRAM comprise about 80% of the total area Figure 6.13(c). In *SAGAR*, simple bypass links are used to achieve the flexibility, which saves about 30% of the area in comparison. From Figure 6.13(d), we observe that NoC is SIGMA consumes about 1.8× more power than *SAGAR*, with NoC consuming 45% of total power.

Performance Comparisons. We analytically model the performance of SIGMA [17], estimating the time taken to stream, compute, and add partial sums as per the functionality described in their paper. In Figure 6.14(a) we plot the simulated runtimes for *SAGAR*, monolithic baseline, and SIGMA with equal number of MAC units (denoted as SIGMA_C)

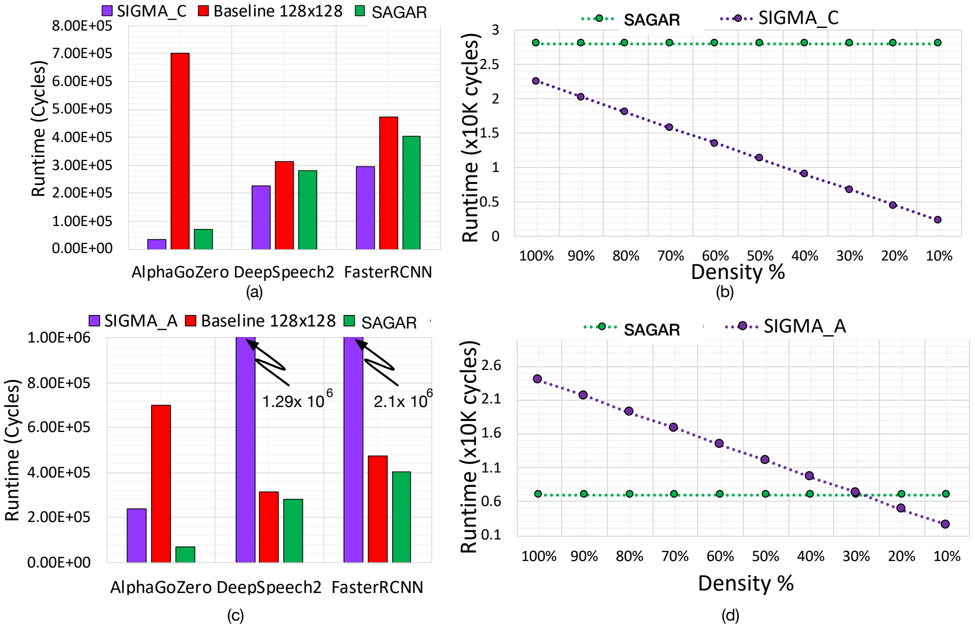


Figure 6.14: Runtimes obtained for (a) running dense workloads for monolithic baseline, *SAGAR*, and *compute normalized* configuration of Sigma (SIGMA_C), (b) *SAGAR* and SIGMA_C configuration by increasing levels of sparsity (decreasing density) in DeepSpeech2, (c) dense workloads for monolithic baseline, *SAGAR*, and *area normalized* configuration of Sigma (SIGMA_A); and (d) *SAGAR* and SIGMA_A configuration by varying levels of sparsity in AlphaGoZero

for our representative workloads. SIGMA_C outperforms *SAGAR* in all workloads. This is due to the fact that the operands are directly streamed to the multiplier over the heavy Benes network, whereas in *SAGAR*, the store-and-forward operation takes up some cycles. The gap in performance further widens with the increase in sparsity as shown in Figure 6.14(b).

As SIGMA implementation takes more area than *SAGAR*, we also compare against the area normalized configuration of SIGMA (denoted as SIGMA_A in Figure 6.14) for fairness. In this case, SIGMA_A consumes about an order of magnitude more number of cycles for each workload as compared to compute normalized configuration, therefore rendering *SAGAR* as the best performer (Figure 6.14(c)). Even when considering workloads with sparse operands, SIGMA_A is able to surpass *SAGAR* only at operand sparsity values above 70% (see Figure 6.14(d)).

6.7 Related Works

Flexible DNN Accelerator. Table 6.1 depicts the standing of various such accelerators in term of native operation supported, mapping capability and flexibility. To efficiently execute a variety of workloads, DNN accelerator designs generally come with two tiers of flexibility, architecture and dataflow. Designs like Neurocube[73], Flexflow[10], and by FPGA based designs[11] enable flexible mapping by supporting multiple dataflow. On the other hand proposals like Planaria [75], Brainwave[8], SIGMA [17], MAERI[9], Cascades[74] and others [11, 12, 20] enable reconfiguration at the hardware level. *RSA* enables both mapping flexibility and reconfigurability.

Dataflow and Accelerator Design Space Search. Several architecture and mapping space exploration tools have been proposed in the recent past to take advantage of flexibilities in the design. Tool like SCALE-Sim[59], MAESTRO[26], Tetris[68] etc. provide analytical models for fast cost estimation of specific configurations. While Timeloop[69], dMazeRunner[28] etc are tools which perform heuristic or exhaustive search for architecture configuration or mapping strategy. SARA systems like *SAGAR* on the other hand use a trained recommender like AdaptNet to circumvent the search and obtain the optimal configuration and dataflow in one shot at runtime.

ML assisted system configuration. Recent works have demonstrated the use of ML algorithms to assist in system configuration. Gamma[34] and ConfuciuX[35] perform architecture mapping and design space configuration search using genetic algorithm and reinforcement learning (RL). On more systems size, work by Mirhoseni et al[37] use RL for task placement on a heterogeneous system, while modern compilers like AutoTVM[36] use ML models for cost prediction to improve compilation time. Nautilus[40] uses genetic algorithm to improve FPGA place and route. It is worth noting that these approaches mostly enhance search for the optimal configuration, and this unlike AdaptNet do not replace search. Perhaps the closest to our approach is work by Kwon et al[41], who use online tensor-based recommender systems to aid place and route in chip design.

6.8 Chapter Summary

In this chapter we demonstrate creation of scalable and flexible DNN accelerator architecture consolidating the designs and techniques discussed in the previous chapters. Specifically, the chapter introduces Self Adaptive Reconfigurable Architectures (SARA), which is class of reconfigurable accelerators aided with a learnt model that can predict the best configuration for a given workload at runtime. This chapter also describes SAGAR, an instance of SARA which is a 32.7 TOPs flexible and scalable reconfigurable DNN accelerator, coupled with AdaptNet recommendation model.

CHAPTER 7
CONCLUSIONS

7.1 Overview of Insights

The work presented in this thesis outlines the tools, methodologies, and reference architectures to design flexible architectures to extract both performance and energy efficiency in the face of every changing workload demands at scale. The following paragraphs summarise the contributions described in the previous chapters in the thesis.

Systematic analysis of scaling DNN accelerators. In Chapter 3, I have described and analytical model and a cycle accurate simulator called SCALE-Sim, which together facilitate fast but detailed analysis of the performance and efficiency of systolic array based DNN accelerators when running different workloads and mapping strategies. These tools are then used to perform an analysis of for scaling strategies for systolic array based accelerators. The main insight of this study are as follows:

- *For systolic based accelerators, the scaling up strategy or building large monolithic computing structures is economical given the regular shape of the array and simple constructions. Provided that workloads can be mapped such that the entire array is utilized; such large arrays are capable of attaining high energy efficiency by extracting high wire reuse. However, the simple and rigid structure of the array makes it extremely difficult to utilize the full computational power of such arrays hindering performance. Alternatively, the scaling out strategy, which involves using multiple computational arrays to collaboratively solve a problem have much higher flexibility in mapping compute and therefore almost always out performs a scaled up monolithic array structure.*
- *Energy efficiency is another important metric to be mindful about when designing*

custom architectures. When scaling systolic array based accelerators, opting for a purely scaled out configuration leads to a loss in operand reuse. Given that any GEMM operation has deterministic work to be done, energy consumption is proportional to the time hardware is kept on to perform the required computation. Lower utilization, leads to higher runtimes, which in turn hurts the energy efficiency. The most efficient configuration therefore is a trade off between giving up operand reuse to gain mapping flexibility. The sweet spot of choosing the right partitioning granularity consequently depends on individual computational workloads. Empirically however, as more compute elements are employed more mapping flexibility is required, making finer grained distributed configurations more desirable.

- *The above two observations dictate that as the workloads increasingly demands high computational performance, future accelerator architectures are required to be flexible if both performance and energy efficiency needs to be preserved.*

High performance flexible accelerator design on FPGA. Chapter 4 describes an implementation of a flexible DNN accelerator on state of the art Xilinx FPGA. The design is motivated by the observations in the previous chapter and aims to exploit flexibility to co-optimize for performance and energy efficiency. FPGAs are a natural implementation choice for flexible architectures. However a serious drawback of FPGA based design is the designs implemented in the programmable fabric struggle to achieve higher clock speeds and barely attain the maximum possible frequency F_{max}. The following point summarizes the novelty and efficacy of the proposed design:

- *The design uses the DSP48 cascades as computational units instead of implementing MAC compute elements on the FPGA fabric. These DSP48 blocks are hard macros implemented on silicon as opposed to any logic emulated using the look up tables (LUTs) on the FPGA fabric, and therefore are capable of running at a high frequency of* $750MHz(F_{max})$ *of Xilinx VU37P Ultrascale+ FPGA.*

- *The DSP48 units are connected using dedicated hardwired interconnects in a daisy chained fashion (cascades) and thus are capable of short distance low latency transfer of operand and results to each other. These compute elements are backed by URAM 288 and RAMB18 (block RAM) structures, which also have their dedicated cascade connections allowing both compute operations and memory transactions to take place at maximum frequency (650 MHz for URAM).*

- *The proposed design leverages the programability of the DSP48 blocks and cascade connections and implements two main type of vectorized compute units which are suitable for convolutions and matrix vector operations respectively. Each of the compute units are designed to extract maximum possible operand and partial sum reuse by exploiting the cascade connections between different DSP48 units or the memory units. Depending upon the workload, the fraction of area dedicated to convolution and matrix vector compute units can be determined at the compile time.*

- *As described in detail in Section 4.5, this design attains an operating frequency of 650 MHz (F_{max} for URAMs) and the reconfiguration allows us to efficiently divide the available space for matrix-vector and convolutions units thus achieving maximum possible energy efficiency.*

Learning design and mapping space of custom architectures. In the next chapter (Chapter 5) we study the design space of systolic array based accelerators to determine if the space can be captured by a machine learning model such that the design optimization task can be automated. In my specific study, three representative yet significant case studies are considered. The first case study deals with finding the optimal array dimensions and mapping (dataflow) for given GEMM operations, subjected to design constraints in terms of maximum computational capacity. The next case study deals with finding the optimal memory sizing for various SRAM buffers feeding the systolic array based accelerator. The final case study optimizes the placement of various workloads onto a collection of heterogeneous accelerators such that runtime is minimized. The following points capture the main observations of this work:

- *The analysis of design space for the three case studies reveal consistent patterns which hold across the changing landscape of input parameters. A couple of insights are apparent from the analyzed data. First, the optimal values in general are clustered in the design space. This renders the learning model naturally to a classification setting, where the hyper planes separating the different clusters are needed to be learnt. Second, the patterns however are only recognizable at higher dimensional spaces with the input parameters as basis.*

- *When choosing the right model for learning the hyperspaces, classifiers appears to be the natural candidate. A classifier based learning framework however requires a separate model for each architecture or mapping parameter to be optimized. In contrast recommendation system models are capable of co-optimizing multiple parameters using a single model. In this work we therefore build a custom recommendation model, which is a simple concatenation of embedding tables for input parameters followed by a simple 2 layer MLP based classifier.*

- *On training the custom designed recommender network on training datasets for the three case studies, we observe that the network learns the optimization space to varying degrees of test accuracy. In comparison to off-the-shelf network topologies, the custom model performs better for each of the cases. Among the various case studies however, we observe that prediction accuracy is usually higher for the ones which have relatively uniform probability distribution across the different output labels*

Self Adaptive Reconfigurable Arrays (SARA). Internalizing the insights obtained from the previous chapters, Chapter 6 presents a architecture which is capable of extracting both high performance and energy efficiency at scale with negligible interference from the software stack. The key motivating factor of the design is that in order to remain performant and energy efficient at scale, the architecture needs to morph between varying degrees of partitioned and monolithic configurations. Furthermore, even when working with a fixed number of partitions the shape of the arrays along with the optimal mapping strategy change when considering different workloads. However, with increased flexibility at scale the reconfiguration space also increases significantly, which poses a challenge for deploying such accelerators. This chapter also describes a method for rapid design and mapping optimization using learnt model. The following points capture the main insights of the work:

- *The architecture comprises of two main sections. The first part is a reconfigurable array (RA) which is morphable in to varying partitioned or monolithic form factors which ever is suited for the best possible execution of the target workload. The second part is a self adaptive (SA) engine, which is the optimizer responsible for configuring the RA part depending on the workload parameters without any optimization input from the software stack.*

- *The RA portion in this proposal is implemented as an array of systolic cells connected together using peer to peer and bypass links. The systolic array like structure enable simple design and high scalability, and also allows to exploit maximum possible operand reuse. The bypass links allow the various systolic cells to work on operands other then the ones obtained from the peer to peer links. The availability of these bypass paths allows the array to be operated as a collection of distributed systolic arrays. The bypass paths can also be disabled and systolic cells can be configured to merged together to work as a larger array whenever needed.*

- *The SA portion of the proposed design comprises of a learnt recommendation model similar to the ones presented in Chapter 5 called AdaptNet. The recommendation model is trained to predict the optimal configuration of the array when queried with dimensions of the GEMM workloads (M, N, and K). The SA portion also comprises of a* *custom hardware block called the AdaptNet-X, which is designed for efficient execution of AdaptNet. Querying the learnt model leads to constant time reconfiguration of the array, which takes about 700 cycles from obtaining the workload parameters to setting the configuration muxes, making it extremely convenient to hide the reconfiguration latency behind the execution of previous workload.*

- *The results presented in Section 6.6 depict that our reference implementation saves about 75% of area and power while providing the same mapping flexibility than that of a comparable fully distributed systolic array based accelerator system. When comparing the energy delay product with baseline monolithic and distributed systolic array configurations with comparable compute capability, the proposed accelerator achieves orders of magnitude better energy efficiency on popular convolution and fully connected layer workloads.*

7.2 Discussions

The data science based analysis of architecture design presented in this thesis opens up a new direction of architecture design and optimization which hopefully opens up a new research efforts. The ultimate goal of the this effort is to build systems which (a) automatically learn the space of optimal design choices, relevant for a variety of workloads and (b) capable of recommending novel design components and system organizations to adapt to changing workload demands. From the analysis depicted in Chapter 5 we can identify a handful of broad areas exploring which will help maturing this direction of research.

- **Systematic analysis of optimization space.** In Section 5.3, we observe that systematic exploration of design space reveals learnable patterns which is exploited by ML models for prediction of optimal architecture or mapping parameters. However, an important observation is that these patterns are manifested in the space of optimal architectural parameters across various workloads and design constraints. This is stark contrast to previous works of using learnt models to assist architecture design, which only aimed to approximate the cost function to speed up search. The discovery of the new space of optimal parameters enables us to directly predict the optimal architecture and mapping configurations. Naturally the next step is to identify and systematically analyze even higher dimensional spaces to help with further generalization on the architecture mapping optimization task.

- **Model construction.** In this work we find that designing a custom recommendation model can not only improve the learning performance, but also allows us to find the optimal values of multiple parameters simultaneously using a single inference pass. The recommendation model however may not be able to capture design spaces beyond a certain threshold of complexity. The next natural question to ask is, weather there exist some novel machine learning model structure which improves on the recommendation model setting in terms of prediction performance, scaling to broader configuration spaces and other capabilities.

- **Learned abstractions.** The custom designed machine learning model described in Section 5.5 shows that multiple architecture and mapping parameters can be combined and optimized simultaneously. However, it not clear if there is limit to the nature and the number of parameters which can be combined and optimized. An ambitious but proportionally impactful study would be the one which can determine the right mix of parameters. Moreover, if it turns out that parameters can be co-optimized across the computation stack then there is a possibility that complex systems that are impractical in the present moment will become much more practical.

7.3 Future Work

Architecture design has always been a data driven decision making process. However, the efforts for automating this process has not come into the mainstream. As of the time of writing this thesis most of design optimizations are done using search based methods, and are performed every time a new optima is needed to be found. There have been a few attempts in the past few years to use learnt models in design space exploration, however these methods almost exclusively focused on learning the cost function to bypass costly simulation steps rather than aiming to replace the iterative search based methods.

In this thesis I propose that learning the design space across different parameters has the potential to drastically reduce the cost of architecture optimization and design. Although the results presented in the previous chapters of this thesis are promising, it is undeniable that the study presented here is just scratching the surface. In the following paragraphs, I present a few immediate and a few long term studies, which I believe will be able to automate architecture design process. Such a goal if achieved with augment the skills of an architect to explore more challenging problems.

1. **Near Term.** Exploring the limits of the recommendation model to capture the optimization space. In the scope of this thesis, the recommendation models presented in Chapter 5, Chapter 6 is capable of capturing the entire optimization space. The results also show that optimization goals involving multiple parameters is also possible to be captured by the same model. However several follow on questions are still needed to be answered. First, if there is a limit to the number of parameters that can be consolidated into a single model? Second, how big can be the output space which can be faithfully classified by a recommender like system? Third, are there similar scalability limitations on the input space size?

2. **Mid Term.** The scope of the studies presented in this thesis focuses on optimizing within the space of a fixed architecture setting, which is our case is a systolic array.

The next natural step is to study if a model can be learnt such that it is capable of recommending different architectural motifs suitable for different kinds of workloads. For example, if there is a choice to chose from a monolithic systolic array, a SIMD array, a CPU or any other systolic accelerator, if it is at all possible for a model to recommend the optimal motif and the corresponding optimal architecture and mapping configurations to run the workloads.

3. **Long Term.** The goals mentioned above only concern with the recommendation space of pre-existing architecture structures. The long term goal of this study is to create a model which can learn to generate the optimal architecture structures, and possibly also create a system by interfacing it with appropriate memory and peripheral structures.

Appendices

APPENDIX A
GENESYS: ENABLING CONTINUOUS LEARNING THROUGH NEURAL NETWORK EVOLUTION IN HARDWARE

A.1 Introduction

Ever since modern computers were invented, the dream of creating an intelligent entity has captivated humanity. We are fortunate to live in an era when, thanks to deep learning, computer programs have paralleled, or in some cases even surpassed human level accuracy in tasks like visual perception or speech synthesis. However, in reality, despite being equipped with powerful algorithms and computers, we are still far away from realizing general purpose AI.

The problem lies in the fact that the development of supervised learning based solutions is mostly open loop (Figure A.1(a)). A typical deep learning model is created by hand-tuning the neural network (NN) topology by a team of experts over multiple iterations, often by trial and error. The said topology is then trained over gargantuan amounts of labeled data, often in the order of petabytes, over weeks at a time on high end computing clusters, to obtain a set of weights. The trained model hence obtained is then deployed in the cloud or at the edge over inference accelerators (such as GPUs, FPGAs, or ASICs). Unfortunately, supervised learning as it operates today breaks if one or more of the following occur:

1. Unavailability of structured labeled data
2. Unknown NN topology for the problem
3. Dynamically changing nature of the problem
4. Unavailability of large computing clusters for training.

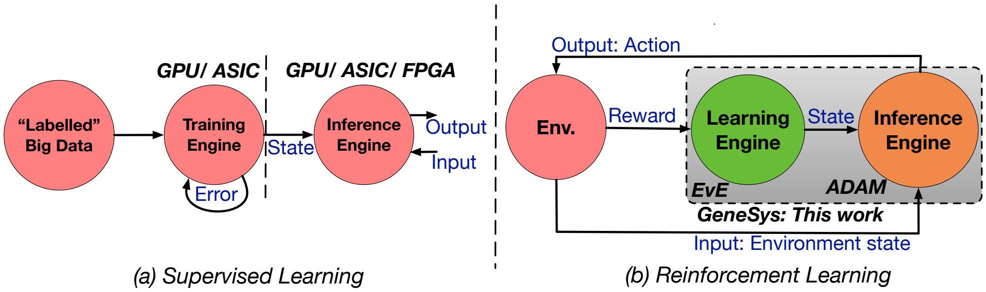


Figure A.1: Conceptual view of GeneSys within machine learning.

Bringing general-purpose AI to autonomous edge devices requires a co-design of the algorithm and architecture for synergistic solution of all four challenges listed above. This chapter presents a work that attempts to solve this problem. We present GeneSys, a system targeted towards energy-efficient acceleration of *neuro-evolutionary (NE) algorithms*. NE algorithms are akin to RL algorithms, but attempt to "evolve" the topology and weights of a NN via genetic algorithms, as shown in Figure A.2. NEs show surprisingly high robustness against the first 3 challenges mentioned earlier, and have seen a resurgence over the past year through work by OpenAI [81], Google Brain [82] and Uber AI Labs [83]. However, these demonstrations have still relied on big compute and memory, which we attempt to solve in this work via clever HW-SW co-design. We make the following contributions:

- We characterize a NE algorithm called NEAT [84], identifying the compute and memory requirements across a suite of environments from OpenAI gym [85].
- We identify opportunities for parallelism (population-level parallelism or PLP and gene-level parallelism or GLP) and data reuse (genome-level reuse or GLR) unique to NE algorithms, providing architects with insights on designing efficient systems for running such algorithms.
- We discuss the key attributes of compute and communication within NE algorithms that makes them inefficient to run on GPUs and other DNN accelerators. We design two novel accelerators, Evolution Engine (EvE) and Accelerator for Dense Addition and Multiplication (ADAM), optimized for running the learning and inference

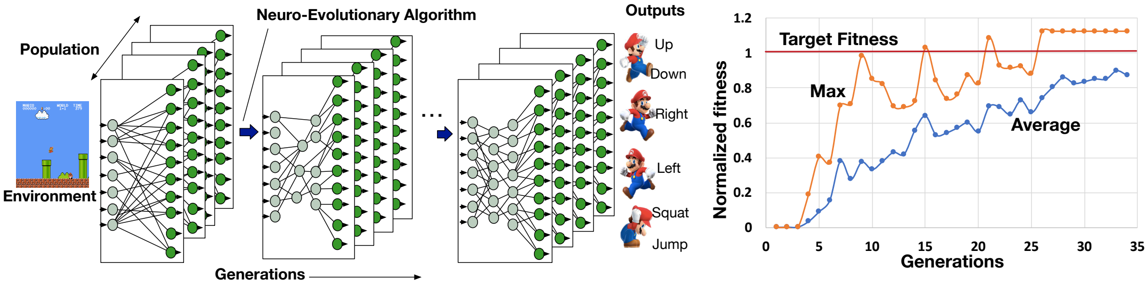
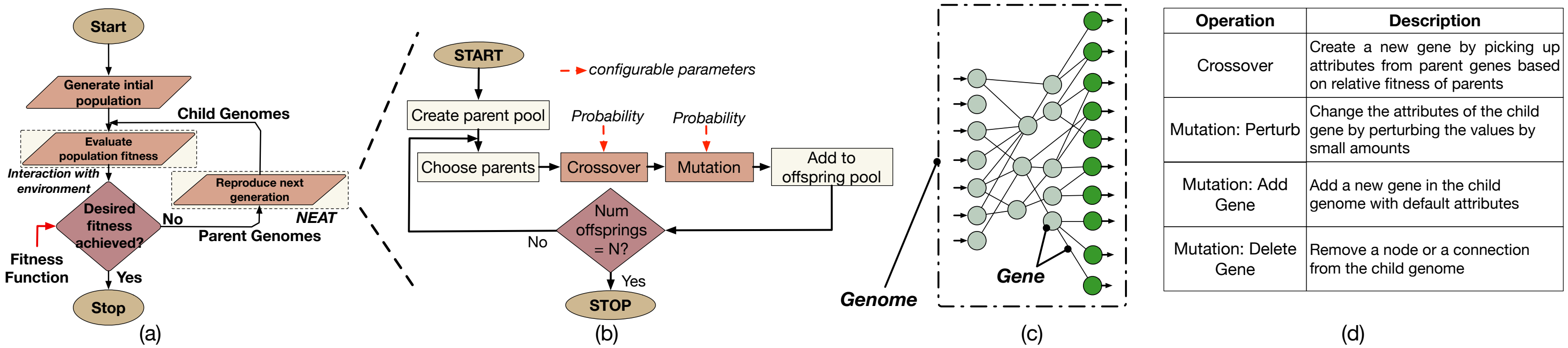


Figure A.2: Example of NE in action, evolving NNs to play Mario.

of NE respectively in hardware, presenting architectural trade-offs along the way. Figure A.1(b) shows an overview.

- We build a GeneSys SoC in 15nm, and evaluate it against optimized NE implementations over latest embedded and desktop CPUs and GPUs. We observe 2-5 orders of magnitude improvement in runtime and energy-efficiency.



Operation	Description
Crossover	Create a new gene by picking up attributes from parent genes based on relative fitness of parents
Mutation: Perturb	Change the attributes of the child gene by perturbing the values by small amounts
Mutation: Add Gene	Add a new gene in the child genome with default attributes
Mutation: Delete Gene	Remove a node or a connection from the child genome

Figure A.3: (a) Flow chart of an evolutionary algorithm (b) The NEAT algorithm (c) Genes and Genomes in context of NNs (d) Ops in NEAT.

A.2 Background

Before we start with the description of our work, we would like to give a brief introduction to some concepts which we hope will help the reader to appreciate the following discussion.

A.2.1 Supervised Learning

Supervised learning is arguably the most widely used learning method used at present. It involves creating a 'policy function" (e.g., a NN topology) (via a process of trial and error by ML researchers) and then running it through tremendous amounts of labelled data. The output of the model is computed for a given set of inputs and compared against an existing label to generate an error value. This error is then backpropogated [86] (BP) via the NN to compute error gradients and update weights. This is done iteratively till convergence is achieved.

Supervised learning has the following limitations as the learning/training engine for general purpose AI:

- Dependence on large structured & labeled datasets to perform effectively without overfitting [87, 88]
- Effectiveness is heavily tied to the NN topology, as we witnessed with deeper *convolution* topologies [56, 46] that led to the birth of Deep Learning.
- Extreme compute and memory requirements [89, 90]. It often takes weeks to train a deep network on a compute cluster consisting of several high end GPUs.

A.2.2 Reinforcement Learning (RL)

Reinforcement learning is used when the structure of the underlying policy function is not known. For instance, suppose we have a a robot learning how to walk. The system has a finite set of outputs (say which leg to move when and in what direction), and the aim is to learn the right policy function so that the robot moves closer to its target destination. Starting with some random initialization, the agent performs a set of actions, and receives an reward from the environment for each of them, which is a metric for success or failure for the given goal. The goal of the RL algorithm is to update its policy such that future reward could be maximized. This is done by iteratively perturbing the actions and computing the corresponding update to the NN parameters via BP. RL algorithms can learn in environments with scarce datasets and without any assumption on the underlying NN topology, but the reliance on BP makes them computationally very expensive.

A.2.3 Evolutionary Algorithms (EA)

Evolutionary algorithms get their name from biological evolution, since at an abstract level they be seen as sampling a population of individuals and allowing the successful individuals to determine the constitution of future generations. Figure A.3(a) illustrates the flow. The algorithm starts with a pool of individuals/agents, each one of which independently tries to perform some action on the environment to solve the problem. Each individual is then assigned a fitness value, depending upon the effectiveness of the action(s) taken by them. Similar to biological systems, each individual is called a *genome*, and is represented by a list of parameters called *genes* that each encode a particular characteristic of the individual. After the fitness calculation is done for all, next generation of individuals are created by crossing over and mutating the genomes of the parents. This step is called reproduction and only a few individuals, with highest fitness values are chosen to act as parents in-order to ensure that only the fittest genes are passed into the next generation. These steps are repeated multiple times until some completion criteria is met.

Mathematically EAs can be viewed as a class of black-box stochastic optimization techniques [81, 83]. The reason they are "black-box" is because they do not make any assumptions about the structure of the underlying function being optimized, they can only evaluate it (like a lookup function). This leads to the fundamental difference between RL and EA. Both try to optimize the expected reward, but RL perturbs the action space and uses backpropagation (which is computation and memory heavy) to compute parameter updates, while EA perturbs the parameter space (e.g., nodes and connections inside a NN) directly. The "black-box" property makes EAs highly robust - the same algorithm can learn how to solve various problems as from the algorithm's perspective the task in hand remains the same: perturb the parameters to maximize reward.

A.2.4 The NEAT Algorithm

TWEANNS are a class of EAs which evolve both the topology and weights for given NN simultaneously. *Neuro-Evolution for Augmented Topologies* (NEAT) is one of the algorithms in this class developed by Stanley et al [84]. We use NEAT to drive the system architecture of GeneSys in this work, though it can be extended to work with other TWEANNs as well. Figure A.3(b) depicts the steps and flow of the NEAT algorithm, and Figure A.3(d) lists the terminology we will use throughout this text.

Population. The population in NEAT is the set of NN topologies in every generation that each run in the environment to collect a fitness score.

Genes. The basic building block in NEAT is a *gene*, which can represent either a NN node (i.e., neuron), or a connection (i.e., synapse), as shown in Figure A.3(c). Each node gene can uniquely be described by an id, the nature of activation (e.g., ReLU) and the bias associated with it. Each connection can be described by its starting and end nodes, and its hyper-parameters (such as weight, enable).

Genome. A collection of genes that uniquely describes one NN in the population, as Figure A.3(c) highlights.

Initialization. NEAT starts with a initial population of very simple topologies comprising only the input and the output layer. It evolves into more complex and sophisticated topologies using the *mutation* and *crossover* functions.

Mutation. Akin to biological operation, mutation is the operation in which a child gene is generated by tweaking the parameters of the parent gene. For instance, a connection gene can be mutated by modifying the weight parameter of the parent gene. Mutations can also involve addition or deletion of genes, with a certain probability.

Crossover. Crossover is the name of the operation in which a child gene for the next generation is created by cherry picking parameters from two parent genes.

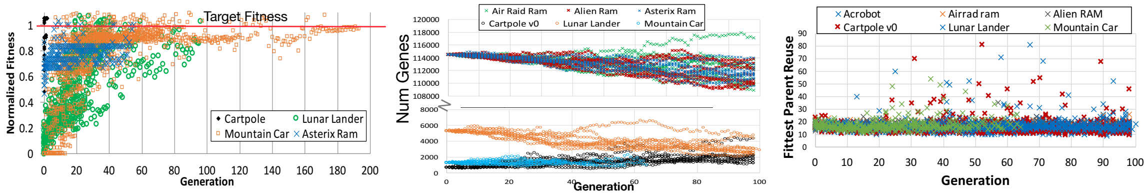
Speciation and Fitness Sharing. Evolutionary algorithms in essence work by pitting the individuals against each other in a given population and competitively selecting the fittest. However, it is not difficult to see that this scheme can prematurely prune individuals with useful topological features just because the new feature has not been optimized yet and hence did not contribute to the fitness. NEAT has two interesting features to counteract that, called *speciation* and *fitness sharing*. Speciation works by grouping a few individuals within the population with a particular niche. Within a species, the fitness of the younger individuals is artificially increased so that they are not obliterated when pitted against older, fitter individuals, thus ensuring that the new innovations are protected for some generations and given enough time to optimize. Fitness sharing is augmenting fitness of young genomes to keep them competitive.

A.3 Computational Behavior of EAs

This section characterizes the computational behavior of EAs, using NEAT as a case study, providing specific insights relevant for computer architects.

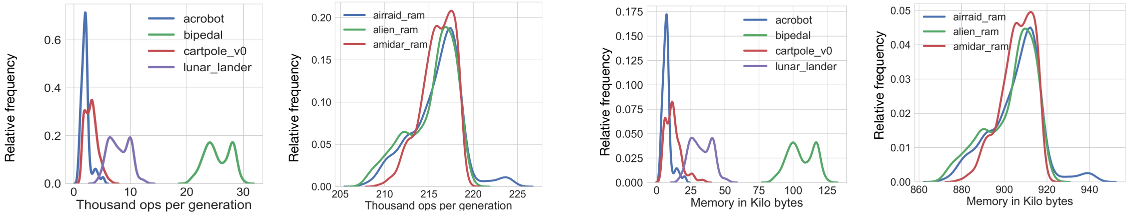
Table A.1: Open AI Gym [85] environments for our experiments.

Environment	**Goal**	**Observation**	**Action**
Acrobot	Balance a complex inverted pendulum constructed by linking two rigid rods	Six floating point numbers	One floating point number
Bipedal	Evolve control for locomotion of a two legged robot on a simple terrain.	Twenty four floating point numbers.	Six floating point numbers.
Cartpole_v0	The winning criteria is to balance an inverted pendulum on a moving platform for 100 consecutive time steps.	Four floating point numbers.	One binary value.
MountainCar	The goal of this task is to control an underpowered car sitting in a valley such that it reaches the finish point on the peak of one of the mountains.	Two floating point numbers.	One integer, less than three, for the direction of motion.
LunarLander	The goal to control the landing of a module to a specific spot on the lunar surface by controlling the fire sequence of its fours thrusters.	Eight floating point numbers.	One integer, less than four, indicating the thruster to fire.
Atari games	The agent has to play Atari games by controlling button presses. We used *Airraid_ram, Alien_ram, Asterix_ram and Amidar_ram* environments	128 bytes indicating the current state of the game RAM.	One integer value, indicating the button press.



(a) Fitness (b) Number of Genes (c) Fittest Parent Reuse

Figure A.4: Evolution behavior of Open AI Gym games as a function of generation.



(a) (b)

Figure A.5: (a) Computation (i.e., Crossover and Mutations) Ops and (b) Memory Footprint of applications from OpenAI Gym in every generation. A distribution is plotted across all generations till convergence and 100 separate runs of each application.

A.3.1 Target Environments

We use a suite of environments described in Table A.1 from OpenAI gym [85]. Each of these environments involves a learning task, which we ran through an open-source python implementation of NEAT [91].

A.3.2 Accuracy and Robustness

All experiments start with the same simple NN topology - a set of input nodes (equal to the observation space of the environment) and a set of output nodes (equal to the action space of the environment). These are fully-connected but the weight on each connection is set to zero. We ran the same code-base for different applications, changing only the fitness function between these different runs. All environments reached the target fitness - demonstrating the robustness of NEAT[1].

[1] We also ran the same environments with open-source implementations of A3C and DQN, two popular RL algorithms, and found that certain OpenAI environments never converged, or required a lot of tuning of the RL parameters for them to converge. However, a comprehensive comparison of RL vs. NE is beyond the scope of this paper.

Figure A.4(a) demonstrates the evolution behavior of four of these environments across multiple runs. We make two observations. First, across environments, there can be variance in the average number of generations it takes to converge. Second, even within the same environment, some runs take longer than others to converge, since the evolution process is probabilistic. For e.g., for *Mountain Car*, the target fitness could be realized as early as generation # 8 to as late as generation # 160. These observations point to the need for energy-efficient hardware to run NE algorithms as their total runtime can vary depending on the specific task they are trying to solve.

A.3.3 Compute Behavior and Parallelism

As shown in Figure A.3(a), EAs essentially comprise of an outer loop running the evolutionary learning algorithm to create new genomes (NNs) every generation, and inner loops performing the inference for these genomes. Prior work has shown that the computation demand of EAs drops by two-thirds compared to backpropagation [81].

Learning (Evolution)

In NEAT, there are primarily two classes of computations that occur - crossover and mutation, as shown in Figure A.3(b). Figure A.5(a) show the distribution of the number of crossover and mutations operations within a generation. The distribution is plotted across all generations till the application converged and across 100 runs of each application. We observe that the mutations and crossovers are in thousands in one class of applications, and are in the range of hundred thousands in another class. A key insight is that *crossover and mutations of each gene can occur in parallel.* This demonstrates a class of raw parallelism provided by EAs that prior work on accelerating EAs [81, 83] has not leveraged. We term this as **gene level parallelism (GLP)** in this work. Moreover, as the environments become more complex with larger NNs with more genes, the amount of GLP actually increases!

Inference

The inference step of NEAT involves running inference through all NNs in the population, for the particular environment at hand. Inference in NEAT however is different than that in traditional Multilevel Level Perceptron (MLP) NNs. Recall that NEAT starts with a simple topology (Section A.3.2) and then adds new connection and nodes via mutation. This way of growing the network results in a irregular topology; or when viewed from the lens of DNN inference - a highly sparse topology. Inference on such topologies is basically processing an acyclic directed graph. An interesting point to note is that, following an evolution step, multiple genomes undergo the inference step concurrently (Figure A.3(a)). As there is no dependence within the genomes, a different opportunity of parallelism arises. We term this as **population level parallelism (PLP)**.

A.3.4 Memory Behavior

Figure A.4(b) plots the total number of genes as the NN evolves.

Memory Footprint.

It is important to note that the memory footprint for EAs at any time is simply the space required to store all the genes of all genomes within a generation. The algorithm does not need to store any state from the previous generations (which effectively gets passed on in the form of children) to perform the learning. From a learning/training point of view, this makes EAs highly attractive - they can have much lower memory footprint than BP, which requires error gradients and datasets from past epochs to be stored in order to run stochastic gradient descent. From an inference point of view, however, the lack of regularity and layer structure means that genomes cannot be encoded as efficiently as convolutionl neural networks today are. There have been other NE algorithms such as HyperNEAT [92] which provide a mechanism to encode the genomes more efficiently, which can be leveraged if need be.

For all the applications in the Open AI gym we looked at, the overall memory footprint per generation was less than 1MB, as Figure A.5(b) shows. While larger applications may have larger memory footprints per generation, the total memory is still expected to be much less than that required by training algorithms due to the reasons mentioned above, enabling a lot of the memory required by the EA to be cached on-chip.

Communication Bandwidth

Leveraging GLP and PLP requires streaming millions of genes to compute units, increasing the memory bandwidth pressure. Caching the necessary genes/genomes on-chip, and leveraging a high-bandwidth network-on-chip (NoC) can help provide this bandwidth, as we demonstrate via GeneSys.

Opportunity for Data Reuse

Data reuse is one of the key techniques used by most accelerators [93]. Unlike DNN inference accelerators which have regular layers like convolutions that directly expose reuse across filter weights, the NN itself is expected to be highly irregular in an evolutionary algorithm. However, we identify a different kind of reuse: **genome level reuse (GLR)**. In every generation, the same fit parent is often used to generate multiple children. We quantify this opportunity in Figure A.5(c). For most applications, the fittest parent in every generation was reused close to 20 times, and for some applications like Cartpole and Lunar lander, this number increased up to 80. In other words, one parent genome was used to generate 80 of the 150 children required in the next generation, offering a tremendous opportunity to read this genome only once from memory and store it locally. This can save both energy and memory bandwidth.

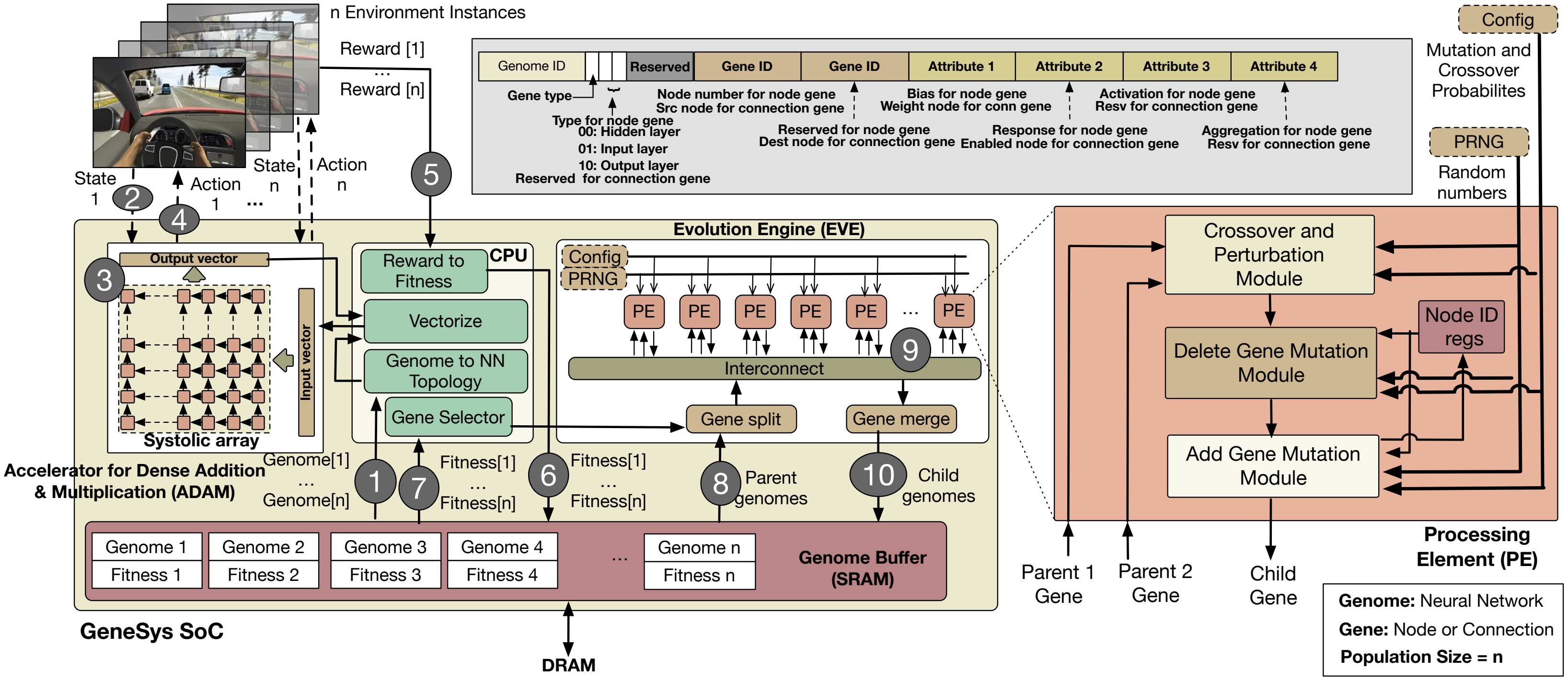


Figure A.6: Overview of the GeneSys system: CPU for algorithm config, Inference Engine (ADAM), Learning Engine (EvE) and EvE PE.

Table A.2: Comparing DQN with EA

	DQN	**EA**
Compute	3M MAC ops in forward pass, 680K gradient calculations in BP	115K MAC ops in inference, 135K crossover + mutations in evolution
Memory	50 MB for replay memory of 100 entries, 4 MB for parameters and activation given mini-batch size of 32	<1MB to fit entire generation
Parallelism	MAC and gradient updates can parallelized per layer	GLP and PLP as described in Section A.3.3 and Section A.3.3
Regularity	Dense CNN with high regularity and opportunity of reuse	Highly sparse and irregular networks

A.3.5 A case for acceleration

In this section we present the key takeaways from the compute and memory analysis of EA. We also compare compute-memory requirements of EA with conventional RL in Table A.2 with DQN [94] as a candidate, both running ATARI.

We notice that EA has both low memory and compute cost when compared to DQN. Given the the reasonable memory foot print (less than 1MB for the applications we looked at) and GLR opportunity, it is evident that a sufficiently sized on chip memory can help remove/reduce off-chip accesses significantly, saving both energy and bandwidth. Also the compute operations in EA (crossover and mutations) are simple and hardware friendly. Furthermore, the absence of gradient calculation and significant communication overheads facilitate scalability [81, 83]. The inference phase of EAs is akin to graph processing or sparse matrix multiplication, and not traditional dense GEMMs like conventional DNNs, dictating the choice of the hardware platform on which they should be run.

If we can reduce the energy consumption of the compute ops by implementing them in hardware, pack a lot of compute engines in a small form factor, and store all the genomes on-chip, complex behaviors can be evolved even in mobile autonomous agents. This is what we seek to do with GeneSys, which we present next.

A.4 Genesys: System and Micro-architecture

A.4.1 System overview

GeneSys is a SoC for running evolutionary algorithms in hardware. This is the first system, to the best of our knowledge, to perform evolutionary learning and inference on the same chip. Figure A.6 present an overview of our design. There are four main components on the SoC:

- *Learning Engine (**EvE**)*: EvE is the accelerator proposed in this work. It is responsible for carrying out the selection and reproduction part of the NEAT algorithm parts of the NEAT algorithm across all genomes of the population. It consists of a collection of processing elements (PEs), designed for power efficient implementation of crossover and mutation operations. Along with the PEs, there is a gene split unit to split the parent genome into individual genes, an on-chip interconnect to send parent genes to the PEs and collect child genes, and a gene merge unit to merge the child genes into a full genome.
- *Inference Engine (**ADAM**)*: We observed in Section A.3.3, the neural nets generated by NEAT are highly irregular in nature. This irregularity deems traditional DNN accelerators unfit for inference in this case, as they are optimized with the assumption that the topology is a dense cascades of layers. In our case inference is closer to graph processing than DNN inference, which is essentially a sequence of multiple vertex updates for the nodes in the NN graph.

  ADAM consists of a systolic array of MAC units to perform parallel vertex evaluations, and a vectorize routine in System CPU to pack nodes into well formed input vectors for dense matrix-vector multiplication. Similar to input vector creation, the vectorize routine also generates weight matrices for genomes, every time a new generation is spawned. However, as the weight matrices do not change within a given generation, and are reused for multiple inferences, while every new vertex evaluation requires a new input vector.

- *System CPU (ARM Cortex M0 CPU)*: We use an embedded Cortex M0 CPU to perform the configuration steps of the NEAT algorithm (setting the various probabilities, population size, fitness equation, and so on), and manage data conversion and movement between EvE, ADAM and the on-chip SRAM.
- *Genome Buffer (SRAM)*: We use a shared multi-banked SRAM that harbors all the genomes for a given generation and is accessed by both ADAM and EvE. This is backed by DRAM for cases when the genomes do not fit on-chip.

A.4.2 Walkthrough Example

We present a brief walk-through of the execution sequence in the system with the help of Figure A.6 to demonstrate the dataflow through the system. Our system starts with a population of genomes of generation *n* in memory. Through the set of steps described, next, GeneSys evolves the genomes for the next generation *n + 1*.

- **Step 1:** The genomes (i.e., NNs) are read from the genome buffer SRAM and mapped over the MAC units in ADAM.
- **Step 2:** ADAM reads the state of the environment. In our evaluations, the environment is one of the OpenAI gym games (Table A.1).
- **Step 3:** Inference is performed by multiple vertex update operations. Several vertices are simultaneously updated by packing input vertices into a well formed vector in the CPU, followed by matrix-vector multiplication on systolic array. Inference for a given genome is marked as complete once the output vertices are updated.
- **Step 4:** The output activations from *step 3* are translated as actions and fed back to the environment.
- **Step 5:** Steps 2-4 are repeated multiple times until a completion criteria is met. For the OpenAI runs, this was either a success or failure in the task at hand. Following this a

cumulative reward value is obtained from the environment - a proxy for performance of the NN.

- **Step 6:** The reward value is then translated into a fitness value by the CPU thread. The reward depends upon the application/environment. The fitness value is augmented to the genome that was just run in SRAM.
- **Step 7:** Once the fitness values for all individuals in the population are obtained, reproduction for the next generation can now start. In NEAT, only individuals above a certain fitness threshold area are allowed to participate in reproduction. A selector logic running on the CPU takes these factors into account and selects the individuals to act as parents in the next generation.
- **Step 8:** The selected parent genomes are read by EvE. The gene splitting logic curates genes from different parents that will produce the child genome, aligns them, and stream them to the PEs in EvE.
- **Step 9:** The PEs receive the parent genes from the interconnect, perform crossovers and mutations to produce the child genes, and send these genes back to interconnect.
- **Step 10:** The gene merge logic organizes the child genes and produces the entire genome. Then this genome is written back into the genome buffer, overwriting the genomes from the previous generation. As each child genome becomes ready, it can be launched over ADAM once again, repeating the whole process.

The system stops when the CPU detects that the target fitness for that application has been achieved. Steps 1 to 6 can leverage PLP, while steps 8 to 10 can leverage GLP. Step 7 (fittest parent selection) is the only serial step.

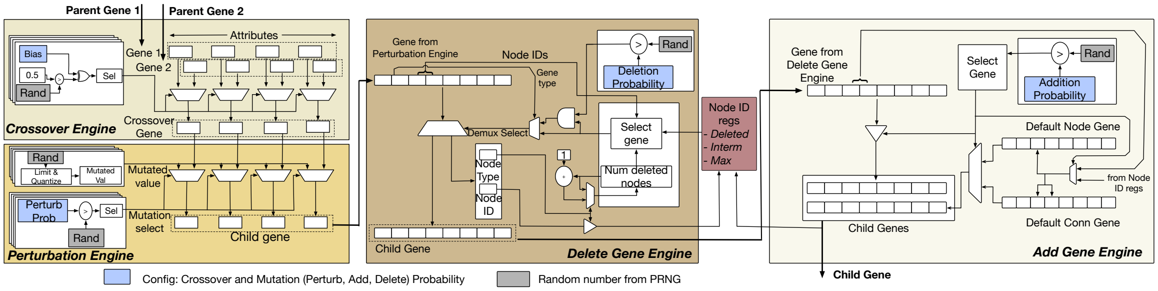


Figure A.7: Schematic depicting the various modules of the Eve PE.

A.4.3 Micro-architecture of EVE

Gene Level Parallelism (GLP)

We leverage parallelism within the evolutionary part - namely at the gene level. As discussed earlier, the operations in an EA can broadly be categorized in two classes: crossover and mutation. In NEAT, there are three kinds of mutations (perturbations, additions and deletions). These four operations are described in Figure A.3(d). While these four operations themselves are serial, they do not have any dependence with other genes. Moreover, the high operation counts per generation (Figure A.5(a)) indicates massive GLP which we exploit in our proposed microarchitecture via multiple PEs.

Gene Encoding

Figure A.6 shows the structure for a gene we use in our design. NEAT uses two types of genes to construct a genome, a node gene which describe vertices and the connection gene which describe the edges in the neural network graph. We use 64 bits to capture both types of genes. Node genes have four attributes - {Bias, Response, Activation, Aggregation} [84]. Connection genes have two attributes - source and destination node ids.

Processing Element (PE)

Figure A.6 shows the schematic of the EvE PE. It has a four-stage pipeline. These stages are shown in Figure A.7. *Perturbation, Delete Gene* and *Add Gene* are three kinds of mutations that our design supports.

Crossover Engine. The crossover engine receives two genes, one from each parent genome. As described in Section A.2.4, crossover requires picking different attributes from the parent genome to construct the child genome. The random number from the PRNG is compared against a *bias* and used to select one of the parents for each of the attributes. We provide the ability to program the bias, depending on which of the two parents contributes more attributes (i.e., is preferred) to the child. The default is 0.5. This logic is replicated for each of the 4 attributes.

Perturbation Engine. A perturbation probability is used to generate a set of mutated values for each of the attributes in the child gene that was generated by the crossover engine.

Delete Gene Engine. There are two types of genes in a given genome - node and connection - and implementing gene deletion for each of them differs. Irrespective of the type, the decision to delete a gene is taken by comparing the deletion probability with a number generated by PRNG. For node deletion, in addition to the probability, the number of previously deleted nodes is also checked. If a threshold amount of nodes are previously deleted, no mode deletion happens in order to keep the genome alive. If not then the node is nullified and its ID is stored. This ID is later compared with the source and destination IDs of any of the connection genes to ensure no dangling connection exist in the genome. Deletion of connections, is fairly straight forward, but deletion decision is taken either by comparing the gene IDs as mentioned above or by comparing deletion probabilities.

Add Gene Engine. This is the fourth and final stage of the PE pipeline. As in the case of the previous stage, depending upon the type of the gene, the implementation varies. To add a new node gene, the logic inserts a new gene with default attributes and a node ID greater than any other node present in the network. Additionally two new connection genes are generated and the incoming connection gene is dropped. The addition of a new connection gene is carried out in two cycles. When a new connection gene arrives, the selection logic compares a random number with the addition probability. If the random number is higher, then the source of the incoming gene is stored. When the next connection gene arrives, the logic reads the destination for that gene, appends the stored source value and default attributes, and creates a new connection gene. This mechanism ensures that any new connection gene that is added by this stage always has valid source and destinations.

Gene Movement

Here, we describe the blocks that manage gene movement.

Gene Selector. As we discussed in Section A.2.4, only a few individuals in a given population get the opportunity to contribute towards the reproduction of the next generation. In very simple terms, selection is performed by determining a fitness threshold and then eliminating the individuals below the threshold. In Section A.2.4 we have seen that NEAT provides a mechanism to keep new features in the population by speciation and fitness sharing. The selection logic in our design works in three steps. First, the fitness values of the individuals in the present generation and read and adjusted to implement fitness sharing. Next, the threshold is calculated using the adjusted fitness values. Finally the parents for the next generation are chosen and the list of parents for the children is forwarded to the gene splitting logic. This is handled by a software thread on the CPU, as shown in Figure A.6.

Gene Split.. The Gene Split block orchestrates the movement of genes from the Genome Buffer to the PEs inside EvE. In the crossover stage, the keys (i.e., node id) for both the parent genes need to be the same. However both the parents need not have the same set of genes or there might be a misalignment between the genes with the same key among the participating parents. The gene split block therefore sits between the PEs and the Genome Buffer to ensure that the alignment is maintained and proper gene pairs are sent to the PEs every cycle.

In addition, this block receives the list of children and their parents from the Gene Selector and takes care of assigning the PEs to generate the child genome. We describe the assignment policy and benefits in Section A.4.3.

Gene Merge. Once a child gene is generated, it is written back to the Gene Memory as part of the larger genome it is part of. This is handled by the Gene Merge block.

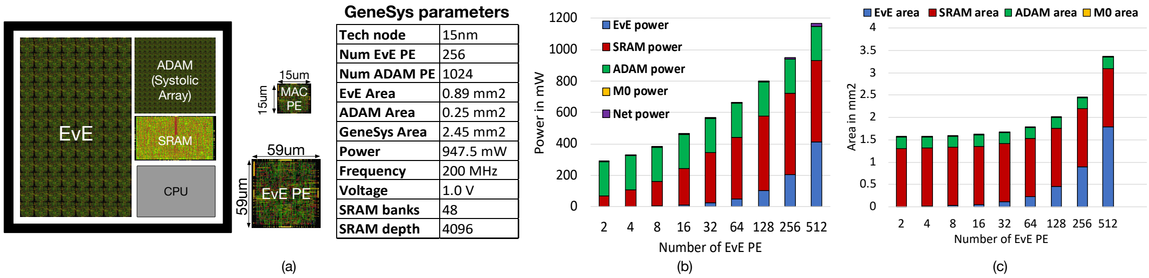
Pseudo Random Number Generators (PRNG). The PRNG feeds a 8-bit random numbers every cycle to all the PEs, as shown in Figure A.6. We use the XOR-WOW algorithm, also used within NVIDIA GPUs, to implement our PRNG.

Network-on-Chip (NoC) A NoC manages the distribution of parent genes from the Gene Split to the PEs and collection of child genes at the Gene Merge. We explored two design options for this network. Our base design is separate high-bandwidth buses, one for the distribution and one for the collection However, recall that the NEAT algorithm offers opportunity for reuse of parent genomes across multiple children, as we showed in Section A.3.4. Thus we also consider a tree-based network with multicast support and evaluate the savings in SRAM reads in Section A.6.

Integration

In this section we will briefly describe how the different components are tied together to build the complete system.

Genome organization. As described in earlier sections, we have two types of genes, nodes and connection. As shown in Figure A.6 each gene can be uniquely identified by the gene IDs. In this implementation we identify node genes with positive integers, and the connection genes by a pair of node IDs representing the source and the destination. Within a genome, the genes are stored in two logical clusters, one for each type. Within each cluster, the genes are stored by sorting them in ascending order of IDs. Ensuring this organization eases up the implementation of the Add Gene engine. During reproduction, since the child gene gets the key of the parent genes, which in turn are streamed in order,



GeneSys parameters

Tech node	15nm
Num EvE PE	256
Num ADAM PE	1024
EvE Area	0.89 mm2
ADAM Area	0.25 mm2
GeneSys Area	2.45 mm2
Power	947.5 mW
Frequency	200 MHz
Voltage	1.0 V
SRAM banks	48
SRAM depth	4096

Figure A.8: (a) Place-and-Routed GeneSys SoC (b) Power consumption with increase in PE in EvE (c) Area footprint with increase in PE in EvE.

ordering is maintained. For newly added genes, the Gene Merge logic ensures that they sequenced in the right order when put together in memory.

EvE Dataflow. After the Gene Selector finalizes the parents and their respective children, the list is passed to the Gene Split block. The Gene Split logic then allocates PEs for generation of the children. In this implementation we allocate only one PE per child genome[2]. The PE allocation is done with a greedy policy, such that maximum number of children can be created from the parents currently in the SRAM. This is done to exploit the reuse opportunity provided by the reproduction algorithm and minimize SRAM reads.

When streaming into the PE, the node genes are streamed first. This is done in order to keep track of the valid node IDs in the genome, which will then be used in the gene addition and deletion mutations. Information about valid nodes are required to prune out dangling connections and assignment of node IDs in case of a new node or connection addition. Once the nodes are streamed, connection genes are streamed until the complete genome of the child is created. Before the genes are streamed, it takes 2 cycles to load the parents' fitness values and other control information.

A.4.4 Microarchitecture of ADAM

As mentioned in Section A.4.1, ADAM evaluates NNs generated by EvE by processing vertices in the irregular NN graph. We had two design choices - either go with a con-

[2] It is possible to spread the genome across multiple PEs as well but might lead to different genes of a genome arriving out-of-order at the Gene Merge block complicating its implementation.

ventional graph accelerator like *Graphicionado* [95], or pack the irregular NN into dense matrix-vector multiplications. Recall that EAs have a small memory requirement (unlike conventional graph workloads) and do not require caching optimizations. Moreover, given that our workloads are neural networks, vertex operations are nothing but multiply and accumulate. We thus decided to go with the latter approach. ADAM performs multiple vertex updates concurrently, by posing the individual vector-vector multiplications into a packed matrix-vector multiplication problem. Systolic array of Multiply and Accumulate (MAC) elements is a well known structure for energy efficient matrix-vector multiplication in hardware, and is essentially the heart of ADAM's microarchitecture.

However, picking the ready node values to create input vectors for packed matrix-vector multiplication is a task with heavy serialization. We use the System CPU to generate required vectors from the node genomes. As both systolic arrays and graph processing are heavily investigated techniques in literature [96, 97, 95, 98, 99, 100], we omit details of implementation for the sake of brevity.

A.5 Implementation

GeneSys SoC. We implemented the GeneSys SoC using Nangate 15nm FreePDK. We implement a 32×32 systolic-array of MAC units for ADAM and measure the post synthesis power and area numbers. EvE PEs are synthesized and the area and power numbers are recorded similar to ADAM, as shown in Figure A.8(a). Figure A.8(b) shows the roofline power as function of EvE PEs. We call it roofline because the numbers here are calculated on the assumption that GeneSys is always computing and thus capture the maximum; actual power will be much lower. In later sections we will discuss why this an overly pessimistic assumption and ways power consumption can be lowered. Motivated by the memory footprint in Section A.3.4, we allocated 1.5MB for on-chip SRAM. The SRAM has 48 banks to exploit the reuse of parents observed in Section A.3.4, as well as to reduce conflict while feeding data to ADAM. We also take into account the area and power contributed

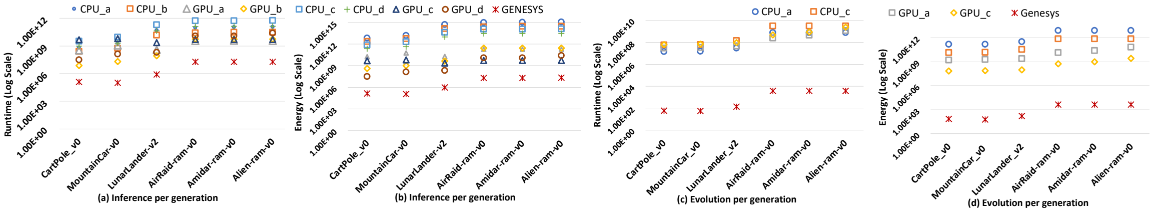


Figure A.9: Runtime and Energy for OpenAI gym environments across CPU, GPU and GeneSys. (a) Runtime, (b) Energy for Inference; and (c) Runtime, (d) Energy of Evolution

by the interconnect and the cortex M0 processor core. With the post synthesis numbers and the relationship of SRAM size and number of PEs, we generate the area footprint for design points with varying number of PE, Figure A.8(c) depicts the numbers.

We choose the operation frequency to be 200MHz, which is typical of the published neural network accelerators [101, 102, 103, 104]. With 256 PEs, we comfortably blanket under 1W as shown in Figure A.8(b).

Table A.3: Target System Configurations.

Legend	Inference	Evolution	Platform
CPU_a	Serial	Serial	6th gen i7
CPU_b	PLP	Serial	6th gen i7
GPU_a	BSP	PLP	Nvidia GTX 1080
GPU_b	BSP + PLP	PLP	Nvidia GTX 1080
CPU_c	Serial	Serial	ARM Cortex A57
CPU_d	PLP	Serial	ARM Cortex A57
GPU_c	BSP	PLP	Nvidia Tegra
GPU_d	BSP + PLP	PLP	Nvidia Tegra
GeneSys	PLP	PLP + GLP	GeneSys

PLP (GLP) - Population (Gene) Level Parallelism
BSP - Bulk Synchronous Parallelism (GPU)

A.6 Evaluation

A.6.1 Methodology

We study the energy, runtime and memory footprint metrics for GeneSys and compare these with the corresponding metrics in embedded and desktop class CPU and GPU platform. For our study we use NEAT python code base [91], and modify the evolution

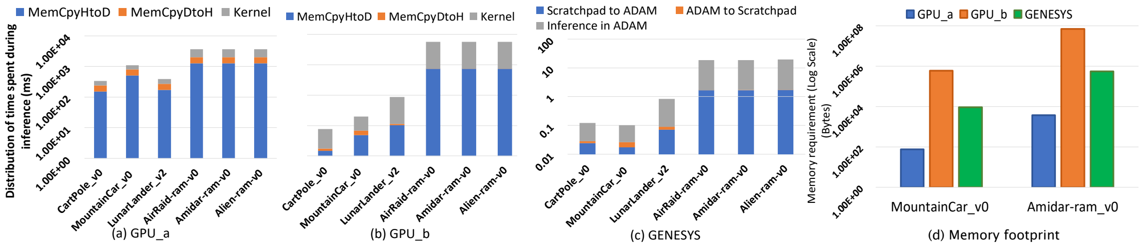


Figure A.10: Distribution of time spent in data-transfer and compute in (a) GPU_a config, (b) GPU_b config and (c) GENESYS; (d) depicts the variation in memory footprints for given application on various platforms

and inference modules as per our needs. We modify the code to optimize for runtime and energy efficiency on GPU and CPU platforms by exploiting parallelism and to generate a trace of reproduction operations for the various workloads presented in Table A.1.

CPU evaluations. We measure the completion time and power measurements on two classes of CPU, desktop and embedded. The desktop CPU is a 6th generation Intel i7, while the embedded CPU is the ARM Cortex A57 housed on Jetson TX2 board. On desktop, power measurements are performed using Intel's power gadget tool while on the Jetson board we use the onboard instrumentation amplifier INA3221. We capture the average runtime for evolution and inference from the codebase, and use it to calculate energy consumption.

GPU evaluations. Similar to CPU measurements, we use desktop (nVidia GTX 1080) and embedded (nVidia Tegra on Jetson TX2) GPU nodes. For the desktop GPU, power is measured using nvidia-smi utility while same onboard INA3221 is used for measuring GPU rail power on TX2. Runtime is captured using nvprof utility for kernels and data-transfers, and are used in energy calculations. To ensure that the correctness of the operations are maintained, we apply some constraints in ordering, for example crossovers precede mutation in time.

GeneSys evaluations. The traces along with the parameters obtained by our analysis in Section A.5 are used to estimate the energy consumption for our chosen design point of EvE. Each line on the trace captures the generation, the child gene and genome id, the type of operation - mutation or crossover, and the parameters changed or added or deleted by the operations. These traces serve as proxy for our workloads when we evaluate EvE and ADAM implementations.

A.6.2 Runtime

Figure A.9(a) and (c) shows the runtime of different OpenAI gym environments on various platforms for both evolution and inference. In CPU, evolution happens sequentially while we try to exploit PLP in inference by using multi-threading, running 4 concurrent threads (CPU_b and CPU_d). Parallel inference on CPU is 3.5 times faster than the serial counterpart.

We try to exploit maximum parallelism in GPU by mapping PLP and GLP to BSP paradigm in inference in two different implementations. Genesys outperforms the best GPU implementation by 100x in inference. Next, we describe our GPU implementations and discuss our observations.

GPU deep dive. GPU_a exploits GLP by forming compaction on input vectors serially and evaluating multiple vertices in parallel for each genome. In GPU_b, multiple vertices across genomes are evaluated in parallel thus exploiting both GLP and PLP. However the inputs and weights could no longer be compacted resulting in large sparse tensors. Figure A.10(a,b,c) depict the contribution of memory transfer in total runtime. We observed memory transfers take 70% of runtime in GPU_a, while GPU_b takes to 20% of total runtime for memory transfer. GeneSys in comparison also take about 15% for memory transfers; however since all the data is on chip, the actual runtime is 1000x smaller. Figure A.10(d) depicts the overall on-chip memory requirement in the GPU_a, GPU_b and GeneSys. We see that GPU_b has a much higher footprint as all sparse weight and input matrices are kept around, while for GPU_a only compact matrices for one genome is required at a time. GeneSys stores entire population in memory, thus we see 100x more footprint than GPU_a, which is expected as we have a population size of 150. GeneSys has 100x less footprint than both GPU_b as GPU_b as genomes rather than sparse-matrices are stored on chip.

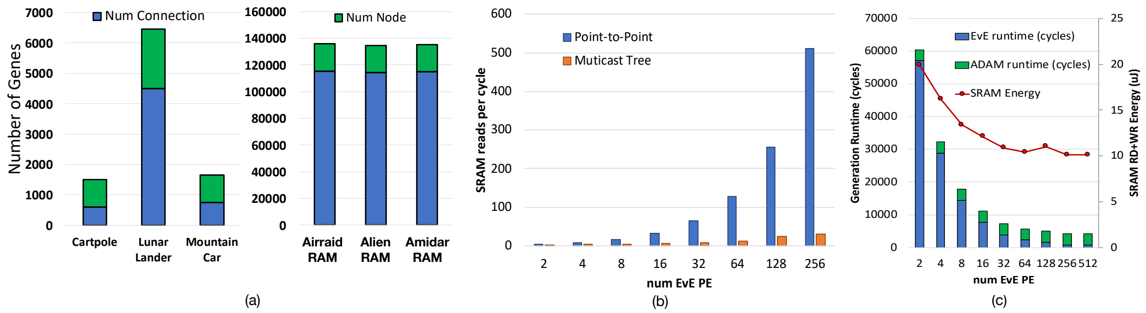


Figure A.11: (a) Composition of gene-types in genomes for different workloads (b) SRAM reads per cycle in Point-to-Point vs Multicast tree (c) SRAM energy consumption and runtime per generation as a function of number of EvE PE averaged for Atari workloads

Figure A.11(a) shows the distribution of connections and nodes in various workloads. The more the number of connection genes means denser weight matrices during inference hence higher utilization in ADAM.

A.6.3 Energy consumption

Figure A.9(b) and (d) shows the energy consumption per generations for OpenAI gym workloads on different platforms. ADAM contributes to 100x more energy efficiency, while EVE turns out to be 4 to 5 orders of magnitude more efficient than GPU_c, the most energy efficient among our platforms.

A.6.4 Design choices: PEs, SRAMs and Interconnect

Impact of Network-on-Chip Neural network accelerators often take advantage of the reuse in data flow to reduce SRAM reads and hence lower the energy consumption. The idea is that, if same data is used in multiple PEs, there is a natural win by reading the data once and multicasting to the consumers. In our case, we see reuse in the parents while producing multiple children of a single parent. Therefore we can use similar methods to reduce reads as well. Figure A.11(b) shows the number of SRAM reads with a simple point-to-point network versus a multicast tree network. We observe more than a 100 × reduction in SRAM reads when supporting multicasts in the network, motivating an intelligent interconnect design. An intelligent interconnect can also help support multiple mapping strategies of genes across the PEs, and is an interesting topic for future research.

Parallelizing Evolution Till now we have talked about EvE PE in terms of *GLP* and reducing compute cost by implementing GA operations in hardware. This line of reasoning can lead to the question that weather *GLP* can be traded-off for energy-benefits. The answer to this lies in Figure A.11(c), where we show the SRAM energy consumption for evolution (Read+Write) and generation time as a function of EvE PEs; size of ADAM and SRAM are constant. The SRAM energy curve indicates that there is almost monotonic improvement in energy efficiency as more EvE PEs are added. The linear decrease in energy (the curve shows exponential decrease for exponential increase in number of PEs) is a direct consequence of *GLR*. At lower PE counts, child genomes sharing same parent PEs are generated over time, thus requiring a single operand to be read over and over again. As the number of PEs increase multiple children sharing the same parent can be serviced by one read if we employ an appropriate interconnect capable of multicasting.

Diverting attention to the runtime plot reveals a couple of interesting trends. First the cycle count for inference is far less than intuitively expected for typical neural networks. This is attributed to two factors, (i) The networks generated by NEAT are significantly simple and small than traditional Deep MLPs, and (ii) ADAM's high throughput aids fast Vector-Matrix computations we use to implement vertex updates. The other more interesting trend that we see is that at lower EvE PE counts the evolution runtime is disproportionately larger than inference! The exponential fall off depicts that performance wise evolution is compute-bound, which is in agreement to our observations on **GLP** and **PLP** in Section A.3.3

Decreasing the generation runtime has further benefits than it meets the eye. In our work we used simulated environments with which we can interact instantly. However, for real life workloads, the interactions will be much slower. This enables us to use circuit level techniques like clock and power gating to save even more power. The lower the compute window for GeneSys the more time is used to interact with the environment thus saving more energy as we hinted in Section A.5.

The tapering off of the trends in Figure A.11(c) at 256 PEs is due to the fact that we exploit only PLP for our experiments and at population size of 150 we intentionally restrict the exploitable parallelism.

A.7 Discussion and Related Work

Future Directions. It is important to note that the success of evolutionary algorithms is tied to the nature of application. From a very high level what EA does, is search for optimal parameters guided by the fitness function and reward value. Naturally, as the parameter space for a problem becomes large, the convergence time of EAs increase as well. In such a scenario, we believe that GeneSys can be run in conjunction with supervised learning, with the former enabling rapid topology exploration and then using conventional training to tune the weights. Neuro-evolution to generate deep neural networks [105, 106, 107, 82, 108, 109] falls in this category. The only thing that would change is the definition of *gene*.

Neuro-evolution. Research on EAs has been ongoing for several decades. [110, 111, 112, 113] are some examples of early works in using evolutionary techniques for topology generations. Apart from NEAT [84], other algorithms like Hyper-NEAT and CPPN [92, 114] for evolution of NNs have also been reported in the last decade [115, 116, 117].

Online Learning. Traditional reinforcement learning methods have also gained traction in the last year with Google announcing AutoML [118, 119, 120]. In situ learning from the environment has also been approached from the direction of spiking neural nets (SNN) [121, 122, 123]. Recently intel released a SNN based online learning chip Loihi [124]. IBM's TrueNorth is also a SNN chip. SNNs have however not managed to demonstrate accuracy across complex learning tasks.

DNN Acceleration. Hardware acceleration of neural networks is a hot research topic with a lot of architecture choices [125, 18, 5, 126, 93, 127, 128, 129, 130] and silicon implementations [101, 102, 103, 104]. These accelerators can replace ADAM for inference, when genes are used to represent layers in MLPs as discussed above. However, EvE remains non-replaceable as there is no hardware platform for efficient evolution in the present to the best of our knowledge.

A.8 Conclusion

This chapter presents GeneSys, an SoC for performing highly energy efficient and performant execution of neuro-evolutionary (NE) algorithms. The implementation of GeneSys in 14m, shows orders of magnitude improvements in energy efficiency and performance when the same algorithm is run on off-the-shelf edge hardware units such as Nvidia Jetson TX2. Apart from the system description, the chapter first characterizes NEAT, a neuro-evolutionary algorithm on several environments in OpenAI Gym. Next, the characterization results are analyzed to determine the feasibility of acceleration and the system design decisions.

REFERENCES

[1] A. Samajdar, P. Mannan, K. Garg, and T. Krishna, "Genesys: Enabling continuous learning through neural network evolution in hardware," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 855–866.

[2] T. Krishna, H. Kwon, A. Parashar, M. Pellauer, and A. Samajdar, "Data orchestration in deep learning accelerators," *Synthesis Lectures on Computer Architecture*, vol. 15, no. 3, pp. 1–164, 2020.

[3] *Nerve Cells and Synapses: Grade 9 Understanding for IGCSE Biology 2.88 2.89*, https://pmgbiology.com/2015/02/18/nerve-cells-and-synapses-a-understanding-for-igcse-biology/, 2015.

[4] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, Salt Lake City, Utah, USA: ACM, 2014, pp. 269–284, ISBN: 978-1-4503-2305-5.

[5] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 43, 2015, pp. 92–104.

[6] Y.-H. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *International Solid-State Circuits Conference*, ser. ISSCC, 2016.

[7] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[8] J. Fowers *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, IEEE Press, 2018, pp. 1–14.

[9] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2018, pp. 461–475.

[10] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, IEEE, 2017, pp. 553–564.

[11] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[12] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–12.

[13] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and W. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA, 2016.

[14] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. S. Emer, S. W. Keckler, and W. J. Dally, "SCNN: an accelerator for compressed-sparse convolutional neural networks," *CoRR*, vol. abs/1708.04485, 2017. arXiv: 1708.04485.

[15] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–12.

[16] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "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)*, IEEE, 2018, pp. 15–28.

[17] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[18] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2014, pp. 609–622.

[19] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, *et al.*, "Simba: Scaling deep-learning inference

with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 14–27.

[20] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, "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*, 2019, pp. 807–820.

[21] K. Rocki, D. Van Essendelft, I. Sharapov, R. Schreiber, M. Morrison, V. Kibardin, A. Portnoy, J. F. Dietiker, M. Syamlal, and M. James, "Fast stencil-code computation on a wafer-scale processor," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2020, pp. 1–14.

[22] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell, *et al.*, "Think fast: A tensor streaming processor (tsp) for accelerating deep learning workloads," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2020, pp. 145–158.

[23] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 97–108, ISBN: 978-1-4799-4394-4.

[24] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, IEEE, 2016, pp. 1–12.

[25] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[26] H. Kwon *et al.*, "Understanding reuse, performance, and hardware cost of dnn dataflows: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2019, pp. 754–768.

[27] P. Chatarasi, H. Kwon, N. Raina, S. Malik, V. Haridas, A. Parashar, M. Pellauer, T. Krishna, and V. Sarkar, "Marvel: A data-centric compiler for dnn operators on spatial accelerators," *arXiv preprint arXiv:2002.07752*, 2020.

[28] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, "Dmazerunner: Executing perfectly nested loops on dataflow accelerators," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–27, 2019.

[29] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, *et al.*, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 369–383.

[30] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Halide: Decoupling algorithms from schedules for high-performance image processing," *Communications of the ACM*, vol. 61, no. 1, pp. 106–115, 2017.

[31] F. Munoz-Martinez, J. L. Abellan, M. E. Acacio, and T. Krishna, "Stonne: A detailed architectural simulator for flexible neural network accelerators," *arXiv preprint arXiv:2006.07137*, 2020.

[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035.

[33] A. Yazdanbakhsh, C. Angermueller, B. Akin, Y. Zhou, A. Jones, M. Hashemi, K. Swersky, S. Chatterjee, R. Narayanaswami, and J. Laudon, "Apollo: Transferable architecture exploration," *arXiv preprint arXiv:2102.01723*, 2021.

[34] S.-C. Kao and T. Krishna, "GAMMA: automating the HW mapping of DNN models on accelerators via genetic algorithm," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, IEEE, 2020, pp. 1–9.

[35] S.-C. Kao, G. Jeong, and T. Krishna, "Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, pp. 622–636.

[36] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Advances in Neural Information Processing Systems*, 2018, pp. 3389–3400.

[37] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," *arXiv preprint arXiv:1706.04972*, 2017.

[38] H. Wang, K. Wang, J. Yang, L. Shen, N. Sun, H.-S. Lee, and S. Han, "Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6.

[39] H. Ren, M. Fojtik, and B. Khailany, "Nvcell: Standard cell layout in advanced technology nodes with reinforcement learning," *arXiv preprint arXiv:2107.07044*, 2021.

[40] M. K. Papamichael, P. Milder, and J. C. Hoe, "Nautilus: Fast automated ip design space search using guided genetic algorithms," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

[41] J. Kwon, M. M. Ziegler, and L. P. Carloni, "A learning-based recommender system for autotuning design fiows of industrial high-performance processors," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2019, pp. 1–6.

[42] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *CoRR*, vol. abs/1704.04760, 2017.

[43] *Xilinx ml suite*, https://github.com/Xilinx/ml-suite, 2018.

[44] H. Kung *et al.*, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2019, pp. 821–834.

[45] P. Rosenfeld *et al.*, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[46] K. He *et al.*, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[47] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[48] D. Amodei *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International Conference on Machine Learning*, 2016, pp. 173–182.

[49] A. Vaswani *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[50] X. He *et al.*, "Neural collaborative filtering," in *Proceedings of the 26th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2017, pp. 173–182.

[51] Xilinx, "Ug570: Ultrascale architecture configuration user guide," 2018.

[52] E. Wu, X. Zhang, D. Berman, and I. Cho, “A high-throughput reconfigurable processing array for neural networks,” in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2017, pp. 1–4.

[53] E. Wu, X. Zhang, D. Berman, I. Cho, and J. Thendean, “Compute-efficient neural-network acceleration,” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 191–200.

[54] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[55] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[56] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[57] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv preprint arXiv:1704.04861*, 2017.

[58] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, “Scale-sim: Systolic CNN Accelerator Simulator,” *arXiv preprint arXiv:1811.02883*, 2018.

[59] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, “A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim,” in *2020 IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS)*, 2020.

[60] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[61] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[62] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[63] xgboost developers, *XGBoost Documentation*, https://xgboost.readthedocs.io/en/latest/index.html, Version 1.3.3.

[64] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," *arXiv*, vol. 1603.04467, 2016.

[65] F. Chollet *et al.*, *Keras*, https://github.com/fchollet/keras, 2015.

[66] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[67] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[68] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 751–764, 2017.

[69] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, IEEE, 2019, pp. 304–315.

[70] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind mappings: Enabling efficient algorithm-accelerator mapping space search," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 943–958.

[71] Z. Zhao, H. Kwon, S. Kuhar, W. Sheng, Z. Mao, and T. Krishna, "Mrna: Enabling efficient mapping space exploration for a reconfiguration neural accelerator," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2019, pp. 282–292.

[72] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[73] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 380–392, 2016.

[74] A. Samajdar, T. Garg, T. Krishna, and N. Kapre, "Scaling the cascades: Interconnect-aware fpga implementation of machine learning problems," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2019, pp. 342–349.

[75] S. Ghodrati, B. H. Ahn, J. K. Kim, S. Kinzer, B. R. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. S. Kim, *et al.*, "Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, pp. 681–697.

[76] T. Krishna, C.-H. O. Chen, W.-C. Kwon, and L.-S. Peh, "Smart: Single-cycle multihop traversals over a shared network on chip," *IEEE micro*, vol. 34, no. 3, pp. 43–56, 2014.

[77] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[78] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Communications of the ACM*, vol. 63, no. 7, pp. 48–57, 2020.

[79] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[80] H. Kwon and T. Krishna, "OpenSMART: Single-cycle multi-hop NoC generator in BSV and Chisel," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2017, pp. 195–204.

[81] T. Salimans, J. Ho, X. Chen, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[82] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, Q. Le, and A. Kurakin, "Large-scale evolution of image classifiers," *arXiv preprint arXiv:1703.01041*, 2017.

[83] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *arXiv preprint arXiv:1712.06567*, 2017.

[84] K. O. Stanley and R. Miikkulainen, "Efficient reinforcement learning through evolving neural network topologies," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, Morgan Kaufmann Publishers Inc., 2002, pp. 569–577.

[85] *Openai gym*, https://github.com/openai/gym, 2017.

[86] D. E. Rumelhart, G. E. Hinton, R. J. Williams, *et al.*, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1,

[87] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[88] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International Journal of Computer Vision*, 2010.

[89] Y. You, A. Buluç, and J. Demmel, "Scaling deep learning on gpu and knights landing clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2017, p. 9.

[90] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "Vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, IEEE, 2016, pp. 1–13.

[91] *Neat python*, https://github.com/CodeReclaimers/neat-python, 2017.

[92] K. O. Stanley, D. D'Ambrosio, and J. Gauci, "A hypercube-based indirect encoding for evolving large-scale neural networks,"

[93] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, 2016, pp. 367–379.

[94] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[95] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Microar-*

chitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on, IEEE, 2016, pp. 1–13.

[96] H. Kung, "Algorithms for vlsi processor arrays," *Introduction to VLSI systems*, pp. 271–292, 1980.

[97] D. I. Moldovan, "On the design of algorithms for vlsi systolic arrays," *Proceedings of the IEEE*, vol. 71, no. 1, pp. 113–120, 1983.

[98] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 105–117, 2016.

[99] G. Dai *et al.*, "Fpgp: Graph processing framework on fpga a case study of breadth-first search," in *FPGA*, ACM, 2016, pp. 105–110.

[100] W.-S. Han *et al.*, "Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc," in *KDD*, ACM, 2013, pp. 77–85.

[101] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 2016, 262–263.

[102] J. Sim, J.-S. Park, M. Kim, D. Bae, Y. Choi, and L.-S. Kim, "14.6 a 1.42 tops/w deep convolutional neural network recognition processor for intelligent ioe systems," in *Solid-State Circuits Conference (ISSCC), 2016 IEEE International*, IEEE, 2016, pp. 264–265.

[103] G. Desoli, N. Chawla, T. Boesch, S.-p. Singh, E. Guidetti, F. De Ambroggi, T. Majo, P. Zambotti, M. Ayodhyawasi, H. Singh, *et al.*, "14.1 a 2.9 tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems," in *Solid-State Circuits Conference (ISSCC), 2017 IEEE International*, IEEE, 2017, pp. 238–239.

[104] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "Envision: A 0.26-to-10 tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 246–257.

[105] J. Bayer, D. Wierstra, J. Togelius, and J. Schmidhuber, "Evolving memory cell structures for sequence learning," *Artificial Neural Networks–ICANN 2009*, pp. 755–764, 2009.

[106] P. Verbancsics and J. Harguess, "Generative neuroevolution for deep learning," *arXiv preprint arXiv:1312.5355*, 2013.

[107] L. Xie and A. Yuille, “Genetic cnn,” *arXiv preprint arXiv:1703.01513*, 2017.

[108] K. Ghazi-Zahedi, “Nmode—neuro-module evolution,” *arXiv preprint arXiv:1701.05121*, 2017.

[109] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, A. Navruzyan, N. Duffy, and B. Hodjat, “Evolving deep neural networks,” *arXiv preprint arXiv:1703.00548*, 2017.

[110] C. M. Taylor, “Selecting neural network topologies: A hybrid approach combining genetic algorithms and neural networks,” *Master of Science, University of Kansas*, 1997.

[111] D. Larkin, A. Kinane, and N. O’Connor, “Towards hardware acceleration of neuroevolution for multimedia processing applications on mobile devices,” in *Neural Information Processing*, Springer, 2006, pp. 1178–1188.

[112] S. Ding, L. Xu, C. Su, and H. Zhu, “Using genetic algorithms to optimize artificial neural networks,” in *Journal of Convergence Information Technology*, Citeseer, 2010.

[113] G. I. Sher, “Dxnn platform: The shedding of biological inefficiencies,” *arXiv preprint arXiv:1011.6022*, 2010.

[114] K. O. Stanley, “Compositional pattern producing networks: A novel abstraction of development,” *Genetic programming and evolvable machines*, vol. 8, no. 2, pp. 131–162, 2007.

[115] D. B. D’Ambrosio and K. O. Stanley, “Generative encoding for multiagent learning,” in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, ACM, 2008, pp. 819–826.

[116] D. Ha, A. Dai, and Q. V. Le, “Hypernetworks,” *arXiv preprint arXiv:1609.09106*, 2016.

[117] C. Fernando, D. Banarse, M. Reynolds, F. Besse, D. Pfau, M. Jaderberg, M. Lanctot, and D. Wierstra, “Convolution by evolution: Differentiable pattern producing networks,” in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, ACM, 2016, pp. 109–116.

[118] *Using machine learning to explore neural network architecture*, https://research.googleblog.com/2017/05/using-machine-learning-to-explore.html, 2017.

[119] B. Zoph and Q. V. Le, “Neural architecture search with reinforcement learning,” *arXiv preprint arXiv:1611.01578*, 2016.

[120] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016.

[121] D. Roggen, S. Hofmann, Y. Thoma, and D. Floreano, "Hardware spiking neural network with run-time reconfigurable connectivity in an autonomous robot," in *Evolvable hardware, 2003. proceedings. nasa/dod conference on*, IEEE, 2003, pp. 189–198.

[122] N. Kasabov, K. Dhoble, N. Nuntalid, and G. Indiveri, "Dynamic evolving spiking neural networks for on-line spatio-and spectro-temporal pattern recognition," *Neural Networks*, vol. 41, pp. 188–201, 2013.

[123] C. D. Schuman, J. S. Plank, A. Disney, and J. Reynolds, "An evolutionary optimization framework for neural networks and neuromorphic architectures," in *Neural Networks (IJCNN), 2016 International Joint Conference on*, IEEE, 2016, pp. 145–154.

[124] *Intel's new self-learning chip promises to accelerate artificial intelligence*, https://newsroom.intel.com/editorials/intels-new-self-learning-chip-promises-accelerate-artificial-intelligence/, 2017.

[125] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014, pp. 269–284.

[126] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015, pp. 161–170.

[127] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ACM, 2017, pp. 27–40.

[128] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *ISCA*, 2016, pp. 1–13.

[129] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *ISCA*, 2016.

[130] J. Kung, D. Kim, and S. Mukhopadhyay, "Dynamic approximation with feedback control for energy-efficient recurrent neural network hardware," in *ISLPED*, 2016, pp. 168–173.

VITA

Ananda Samajdar was born on 9th July 1990 in New Delhi India. He obtained his Bachelors degree from Indian Institution of Information Technology, Allahabad in 2013, majoring in Electronics and Communication Engineering. Ananda moved to Atlanta, GA, USA in 2016 to pursue his PhD in Computer Engineering from Georgia Institute of Technology under Dr. Tushar Krishna's supervision.

Ananda is passionate about exploring computer systems and his research interests lie in Computer Architecture, Systems Design, and Machine Learning techniques. Before joining to Georgia Tech, he worked as VLSI design engineer at Qualcomm Bangalore Design Center. Ananda's work has been published in multiple top tier architecture conferences. He has also a co-author of the book, ”Data Orchestration in Deep Learning Accelerators”.