FPGA-Based Edge-Computing Acceleration

by

Saman Biookaghazadeh

A Dissertation Presented in Partial Fulfillment
of the Requirement for the Degree
Doctor of Philosophy

Approved July 2021 by the
Graduate Supervisory Committee:

Ming Zhao, Co-Chair
Fengbo Ren, Co-Chair
Baoxin Li
Jae-Sun Seo

ARIZONA STATE UNIVERSITY

August 2021

ABSTRACT

The rapid growth of Internet-of-things (IoT) and artificial intelligence applications have called forth a new computing paradigm–edge computing. Edge computing applications, such as video surveillance, autonomous driving, and augmented reality, are highly computationally intensive and require real-time processing. Current edge systems are typically based on commodity general-purpose hardware such as Central Processing Units (CPUs) and Graphical Processing Units (GPUs) , which are mainly designed for large, non-time-sensitive jobs in the cloud and do not match the needs of the edge workloads. Also, these systems are usually power hungry and are not suitable for resource-constrained edge deployments. Such application-hardware mismatch calls forth a new computing backbone to support the high-bandwidth, low-latency, and energy-efficient requirements. Also, the new system should be able to support a variety of edge applications with different characteristics.

This thesis addresses the above challenges by studying the use of Field Programmable Gate Array (FPGA) -based computing systems for accelerating the edge workloads, from three critical angles. First, it investigates the feasibility of FPGAs for edge computing, in comparison to conventional CPUs and GPUs. Second, it studies the acceleration of common algorithmic characteristics, identified as loop patterns, using FPGAs, and develops a benchmark tool for analyzing the performance of these patterns on different accelerators. Third, it designs a new edge computing platform using multiple clustered FPGAs to provide high-bandwidth and low-latency acceleration of convolutional neural networks (CNNs) widely used in edge applications. Finally, it studies the acceleration of the emerging neural networks, randomly-wired neural networks, on the multi-FPGA platform.

The experimental results from this work show that the new generation of workloads requires rethinking the current edge-computing architecture. First, through the

acceleration of common loops, it demonstrates that FPGAs can outperform GPUs in specific loops types up to 14 times. Second, it shows the linear scalability of multi-FPGA platforms in accelerating neural networks. Third, it demonstrates the superiority of the new scheduler to optimally place randomly-wired neural networks on multi-FPGA platforms with 81.1 times better throughput than the available scheduling mechanisms.

This dissertation is dedicated to:

my parents, for they gave me everything unconditionally

my sister and step-brother, who has supported me throughout my long journey

and my true friends, whom always been there, when I needed them.

ACKNOWLEDGEMENTS

I want to express my most profound appreciation to Dr. Ming Zhao for the guidance and help he generously provided me throughout these past few years. I am very thankful for the freedom I was given to explore various research directions and find what interests me most. From the long list of qualities I hope to have acquired from him, above all was his integrity, ethics, deep thinking, and unconditional support.

I want to thank Dr. Fenbo Ren for all his time and efforts throughout my research, especially helping me to understand hardware systems better. He generously assisted me with all the equipment and helped me learn the necessary knowledge. Without his support, I wouldn't have been able to start my dissertation smoothly.

I would also like to thank my committee members: Dr. Baoxin Li and Dr. Jae-Sun Seo, for their open-mindness, insightful bits of advice, and constructive feedback, which assisted me in the completion of this thesis. In addition, I give thanks to the exceptional one-on-one meetings with them, which improved the quality of my research leading to this point.

Furthermore, I would like to acknowledge Dulcardo Artega, Yiqi Xu, Wenji Li, Jorge Cabrera, Gregory Jean-Baptise, Douglas Otstott, Yitao Chen, Qirui Yang, Runyu Jin, and everyone from the VISA Lab. They all helped me learn about research and shape my ideas in the best fashion throughout my journey. I wish all these individuals the best of luck, and I hope our paths cross each other again soon.

Next, I must specially thank Dr. Raju Rangaswami. Throughout my time at Florida International University, Dr. Rangaswami allowed me to learn how academic ideas can transform into technology products. He taught me a lot about advanced storage systems topics throughout this journey and helped me with many new things.

Also, I must thank all my close friends who made my Ph.D. experience much more enjoyable. One great benefit of my Ph.D. experience was finding new friends from

whom I learned a lot. I wish all of them the best in their future endeavors and thank their indirect but significant support for my work.

Last but not least, I would not have been able to complete my dissertation without the help of many others here at ASU, some of whom are unknown to me. That is why I would like to extend my appreciation to the administrative staff, especially Monica Dugan, Pamela Dunn, Christina Sebring, and Jaya Krishnamurthy.

Thank you all!

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Chapter 1

INTRODUCTION

The Internet-of-Things (IoT) will connect 50 billion devices and is expected to generate 400 Zetta Bytes of data per year by 2020. Even considering the fast-growing size of the cloud infrastructure, the cloud is projected to fall short by two orders of magnitude to either transfer, store, or process such vast amount of streaming data Fowers *et al.* (2012). Consequently, the consensus in the industry is to expand our computational infrastructure from data centers towards the edge. Existing edge servers on the market are simply a miniature version of cloud servers (cloudlet) which are primarily structured based on CPUs with tightly coupled co-processors (e.g., GPUs) HPE (2019b,a); Cisco (2019). However, CPUs and GPUs are optimized towards batch processing of in-memory data and can hardly provide consistent nor predictable performance for processing streaming data coming dynamically from I/O channels. Therefore, future edge servers call for a new general-purpose computing system stack tailored for processing streaming data from various I/O channels at low power consumption and high energy efficiency.

FPGAs are a great candidate to address the edge-computing challenges by harnessing the programmability and the ability to handle streaming data. FPGAs can be deployed alongside the conventional accelerators and enable a new generation of heterogeneity for accelerating the different type of IoT application. With the emergence of FPGAs, the benefits of heterogeneous systems become more significant as the FPGAs can handle a specific class of application, in both the cloud and the edge.

While FPGAs are a great candidate for the next generation of the edge systems, there are several challenges the need to be addressed to make effective use of FPGA accelerators in the edge systems:

1.1 FPGA in the Edge

Over the next decade, a vast number of edge servers will be deployed to the proximity of IoT devices; a paradigm that is now referred to as fog/edge computing.

There are fundamental differences between traditional cloud and the emerging edge infrastructure. The cloud infrastructure is mainly designed for (1) fulfilling time-insensitive applications in a centralized environment; (2) serving interactive requests from end users; and (3) processing batches of static data loaded from memory/storage systems. Differently, the emerging edge infrastructure has distinct characteristics, as it keeps the promise for (1) servicing time-sensitive applications in a geographically distributed fashion; (2) mainly serving requests from IoT devices, and (3) processing streams of data from various input/output (I/O) channels. Existing IoT workloads often arrive with considerable variance in data size and require extensive computation, such as in the applications of artificial intelligence, machine learning, and natural language processing. Also, the service requests from IoT devices are usually latency-sensitive. Therefore, having a predictable performance to various workload sizes is critical for edge servers.

Existing edge servers on the market are simply a miniature version of cloud servers (cloudlet) which are primarily structured based on CPUs with tightly coupled co-processors (e.g., GPUs). However, CPUs and GPUs are optimized towards batch processing of in-memory data and can hardly provide consistent nor predictable performance for processing streaming data coming dynamically from I/O channels. Furthermore, CPUs and GPUs are power hungry and have limited energy efficiency [4],

creating enormous difficulties for deploying them in energy- or thermal-constrained application scenarios. Therefore, future edge servers call for a new general-purpose computing system stack tailored for processing streaming data from various I/O channels at low power consumption and high energy efficiency.

OpenCL-based field-programmable gate array (FPGA) computing is a promising technology for addressing the aforementioned challenges. FPGAs are highly energy-efficient and adaptive to a variety of workloads. Additionally, the prevalence of high-level synthesis (HLS) has made them more accessible to existing computing infrastructures.

1.2 Loop Acceleration in Heterogeneous Systems

Many applications can benefit from computing on hardware accelerators, ranging from cloud computing to big-data and edge computing. Examples of these applications include (1) analysis of large quantity of data on big-data platforms, (2) training and running artificial intelligence (AI) and machine learning models in the cloud, (3) processing streams of requests and data from IoT devices, and (4) modeling and simulating the behaviors of scientific applications.

By using accelerators, applications can achieve higher throughput Owens *et al.* (2008), lower response time Biookaghazadeh *et al.* (2018), and/or lower energy consumption Fowers *et al.* (2012).

A variety of accelerators are readily available for applications to choose for their computation needs in the cloud. Graphics Processing Units (GPUs) are the most widely used and can be easily found in many HPC and cloud systems. Other types of accelerators are also becoming increasingly available, e.g., Tensor Processing Units (TPUs) on the Google cloud and Field-Programmable Gate Arrays (FPGAs) on the Amazon cloud (F1 nodes). These accelerators come with different capabilities and limitations. For example, FPGAs can be reconfigured to run any applications but can provide only low clock frequency; GPUs can be programmed using high-level languages to accelerate highly parallel applications; and TPUs are specifically designed for deep learning workloads. Although a general understanding of different accelerators is available, choosing the right accelerators for applications in a heterogeneous computing system is still a difficult problem.

Several related works have studied the performance of common algorithms on accelerators. For example, Rodinia benchmark and its follow-up work Zohouri *et al.* (2016) are designed to benchmark heterogeneous platforms including CPUs, GPU, and FPGAs. These benchmarks usually provide insights on a macro level, for a complete algorithm on a hardware platform. However, they lack a thorough analysis of micro-level execution patterns that exist in different applications and the effectiveness of different hardware architectures in handling these patterns.

To address the above challenges, we study how the accelerators with different hardware architectures can accelerate different types of loops, which are the basic building blocks of almost every computationally intensive application. These applications typically consist of one or many nested and flattened loops. These loops can

embody different patterns in terms of types and degrees of dependency and concurrency, and they can be found in many applications. For example, dynamic programming algorithms consist of one or more nested loops, where every iteration depends on another iteration that points diagonally in the iteration space. Therefore, abstracting the common loop patterns from applications and understanding how they perform on various hardware accelerators are essential steps towards optimally utilizing the accelerators for executing different applications. Although there is a great body of existing works on loop optimizations Wang *et al.* (2021); Juega *et al.* (2014); Konstantinidis *et al.* (2013); Baghdadi *et al.* (2019); Simbürger *et al.* (2013); Trifunovic *et al.* (2010); Grosser *et al.* (2012, 2011); Bastoul (2004); Loechner (1999); Ancourt and Irigoin (1991); Schreiber *et al.* (1990); Cousot and Halbwachs (1978); Lamport (1974), they cannot provide cross-accelerator comparisons that can help developers choose the right platform for their applications in a heterogeneous computing system.

To support the study of loop accelerations across different platforms, we developed *Loopy*, a collection of five fine-grained loop patterns that commonly exist in real-world applications such as linear algebra, optimization, and data analytics algorithms. Loopy parameterizes the key aspects of these loop patterns, including the type and degree of dependencies, data bit-precision, operational intensity, and size of the iteration spaces. It allows them to be flexibly tuned to model diverse loop characteristics. Loopy provides optimized OpenCL implementations of these loop patterns for both GPU and FPGA, the two most versatile and available accelerators. We focus on OpenCL because it is an important framework for the emerging heterogeneous computing paradigm.

Based on Loopy, we evaluated the performance of important loop patterns on several typical accelerators, including Intel A10 FPGAs and Nvidia T4 and RTX2080 GPUs. Our study made several key findings. First, for three out of five loop de-

pendency patterns (intra-dimension dependency, conditional dependency, and half-parallelism half-dependency), FPGA has the potential to outperform GPU. For example, for the intra-dimension dependency pattern, the evaluated FPGA outperforms GPU by 17.5x. Second, for various computational intensities, FPGA can maintain an identical performance, whereas GPU performance is highly variable. For example, having eight conditional statements can degrade the GPU performance by up to 45%. Third, increasing the input data size can increase the performance difference between these two accelerators. For example, for the diagonal dependency loop pattern, the performance gap increases by 51%, while changing the input data size from 4MB to 256MB.

1.3 Multi-FPGA Acceleration of AI on the Edge

In recent years, FPGAs have received tremendous attention in the world of neural network acceleration. FPGAs can provide unique benefits to accelerate Convolutional Neural Networks (CNNs). First, FPGAs can guarantee tight latency bounds for incoming requests. Conventional CNN accelerators, i.e., GPUs, have shown the ability for the acceleration of a batch of requests, by leveraging their farm of processing cores. Unfortunately, they lack the potential to guarantee low-latency services for individual requests Zhang *et al.* (2016, 2018). In contrast to GPUs, FPGAs can leverage their reconfigurable deep pipeline to service the requests in a streaming fashion and provide a predictable low latency. Second, conventional processors are usually power-hungry, which makes them challenging to deploy in power- or energy-constrained environments. Differently, FPGAs are highly power-efficient due to their low operational clock frequency. In conclusion, FPGAs are considered as an excellent platform for accelerating CNNs for deployment.

The ever-increasing complexity of emerging CNNs requires FPGAs with a higher amount of resources, such as memory bandwidth and logical units, to achieve low-latency and high-throughput inferences. Even high-end FPGA chip technologies can host only a small section of a whole CNN model. For example, the Intel Stratix 10 FPGA can perform only 5000 multiply-accumulation (MAC) operations per clock cycle, which is even less than the total number of operations for a single layer of a typical CNN, such as VGG-16 or ResNet. As a result, they fall short in handling heavier CNNs for ultra-low latency (less than ten milliseconds), and high-throughput (more than 60 images/frames per second). Such a problem is even more significant for accelerating more computationally intensive operations, for example, three-dimensional (3D) convolutions, which show great potentials in video processing applications. This challenge can be potentially addressed by utilizing a cluster of FPGAs, connected through a high-bandwidth communication infrastructure.

Achieving linear speedup using a multi-FPGA solution is not straightforward. First, we need to have an efficient design on a single FPGA and achieve state-of-the-art performance. Such performance benefits should be reflected in the acceleration of various CNN operations. Second, the pipeline of multiple FPGAs should be correctly managed to ensure that all FPGAs are doing useful works to handle incoming requests. Third, CNN partitioning, which is the process of mapping different parts of the model onto different FPGAs, should be done intelligently to make sure the workload is balanced across the FPGAs.

Related works Zhang *et al.* (2016); Jiang *et al.* (2019) have studied the multi-FPGA acceleration of neural networks. These works come with several limitations. First, they do not provide a general architecture to accelerate various types of CNNs. For example, they are only able to accelerate either two-dimensional (2D) or 3D convolutions, but not both. Second, they do not optimally exploit the FPGA acceler-

ation resources, which leads to sub-optimal performance, compared to the maximum theoretical performance of an FPGA. Third, they are designed and developed, using low-level hardware programming languages (Jiang et al. Jiang *et al.* (2019) used Xilinx HLS), which makes it difficult to extend and support by the widely-used deep learning frameworks, such as Tensorflow Abadi *et al.* (2016) and Caffe Jia *et al.* (2014).

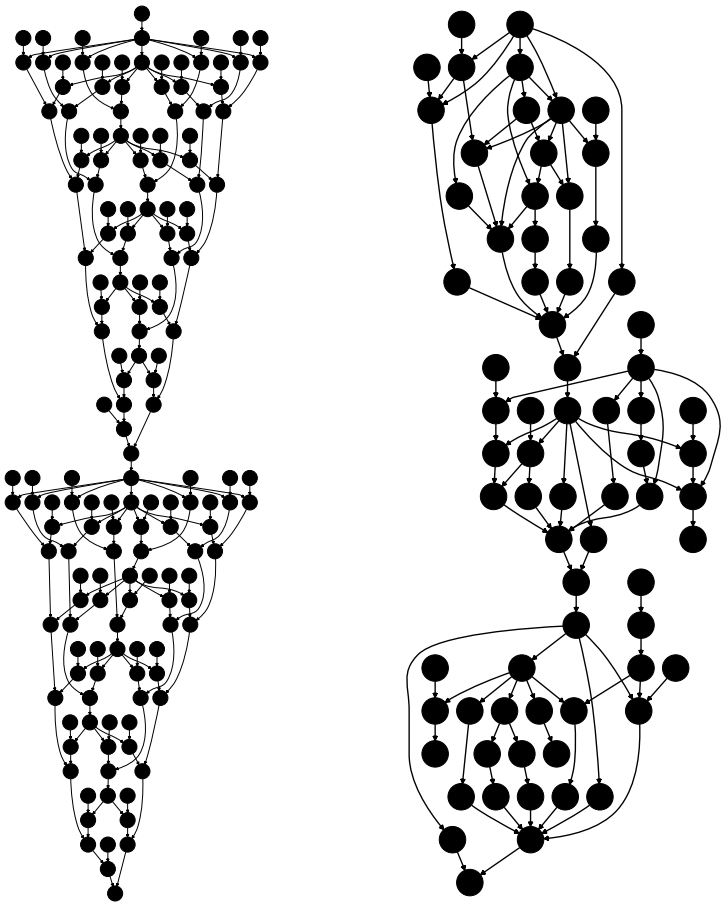
In this thesis, we present a novel multi-FPGA CNN accelerator that can leverage a deep pipeline of FPGAs, connected through a high-performance I/O channel. First, we adopted the Intel Deep Learning Accelerator (DLA) Aydonat *et al.* (2017) architecture and applied various optimizations to achieve an efficient design on a single FPGA. Using a novel systolic array design, our architecture has reduced the total resource consumption of the DLA by up to 25% and increased the overall performance by 24%. We developed this design using OpenCL, which enables convenient integration with widely-used deep learning frameworks. Also, it enables the integration of the accelerator in a heterogeneous environment, where the same OpenCL code can run across different processors. Second, we extended the design to support data communication with other FPGAs in the pipeline, using a 40Gb/s QSFP+ I/O channel. Using a network of connected FPGAs enables temporal (distributing the layers onto different FPGAs) and spatial (splitting a single layer and mapping it onto multiple FPGAs) parallelization of the layers. Using this configuration, a user can allocate a set of FPGAs in a network, with no prior information about the network architecture. The user can interact with these FPGAs as a single FPGA with a large number of resources. Further, she/he can select a neural network model and deploy it on these FPGAs. The framework can automatically split the model into several sub-models, and deploy each sub-model onto an FPGA. This cluster of FPGAs can provide the same or better latency and energy-efficiency, compared to the available CPU or GPU solutions. Third, we extended the design to support 3D convolutions, on top of 2D

convolutions, for certain types of emerging CNN applications. Fourth, we developed a model and strategy for optimizing the partitioning and the placement of the CNN layers on the set of available FPGAs in the pipeline.

To demonstrate the feasibility of our framework, we performed multiple experiments using different widely-used CNN models. Our CNN models for the experiments are VGG-16, Alexnet, and ResNet, which are 2D models commonly used for image classification, and C3D, which is a 3D model commonly used for video processing. We deployed these models on a single- and multi-FPGA pipelines. Our results show that using the multi-FPGA configuration can increase the throughput, almost linearly, with respect to the total number of FPGAs. Also, our extended systolic array shows superior performance (up to 1.7 times), compared to other related works, for accelerating the 3D convolution-based CNN architectures.

1.4 Scheduling Randomly-Wired Neural Networks

Deep neural networks (DNNs) have outperformed many conventional machine learning approaches in the level of accuracy improvement. Hence, DNNs are massively used in various scenarios, such as augmented reality, face recognition, and object classification. State-of-the-art neural networks leverage deeper and wider architectures to improve the accuracy on a larger set of data. Still, they suffer from feasible performance on a wide variety of hardware platforms. A recent body of work, *Neural Architecture Search (NAS)* Zoph and Le (2016); Zoph *et al.* (2018); Liu *et al.* (2018a); Cai *et al.* (2018); Real *et al.* (2019); Cheng *et al.* (2019) and *Random Network Generators* Xie *et al.* (2019); Wortsman *et al.* (2019), propose a distinct architecture model, which promises smaller model size. In these new architecture models, which are generally known as *Randomly-Wired Neural Networks (RWNNs)*, layers can read-/write the input/output from/to any other layer in the RWNN graph. This is in



(a) RandWire (b) SwiftNet

Figure 1.1: *Randomly-wired Neural Networks (RWNNs).*

contrast with traditional models, where layers are structured linearly and read/write directly from the predecessor/successor layers. Need to mention, few emergent works, such as ResNet He *et al.* (2016) and DenseNet Huang *et al.* (2017), have exercised the idea of feeding multiple inputs to a layer but are hand-tuned manually. Such randomness of connections between the layers unlocks a larger space of layer orientations, which helps develop networks with significantly fewer parameters and state-of-the-art accuracy.

Hardware acceleration is another effort to enable large-scale DNN models in pro-

duction Sze *et al.* (2017) efficiently. FPGAs have received tremendous attention in the world of neural network acceleration Biookaghazadeh *et al.* (2020). The benefits of the FPGAs is two-fold. First, they can guarantee tight latency bounds for incoming requests. In other words, FPGAs can leverage their re-configurable deep pipeline to service the requests in a streaming fashion and provide a predictable low latency. Second, FPGAs are highly power-efficient due to their low operational clock frequency compared to other power-hungry processors (CPUs and GPUs). As a result, FPGAs are considered an excellent platform for accelerating DNNs for deployment.

The recent development of multi-FPGA setups enabled higher throughput Fowers *et al.* (2018); Biookaghazadeh *et al.* (2020) or lower latency Jiang *et al.* (2019) inferences through mapping the layers onto multiple FPGAs, which are oriented linearly. For example, Microsoft Brainwave Fowers *et al.* (2018) leverages a network of FPGAs to parallelize and pipeline the execution of the layers at scale. In this setup, the DNN model is being distributed onto the FPGAs by splitting the design, linearly, into N parts, where N is the number of FPGAs. Further, the layer parameters are being pre-loaded onto their respective FPGAs memory. Finally, the input stream of data feeds into the pipeline, and each FPGA performs a portion of the calculation. Doing so enables improving the performance and energy-efficiency of the DNN acceleration on the cloud.

The scheduling methods Biookaghazadeh *et al.* (2018); Chiou (2017) for the available multi-FPGA setups are not designed for RWNNs, due to their complicated connectivity. As a result, RWNNs cannot fully benefit from running on these multi-FPGA accelerators. Extending schedulers Abadi *et al.* (2016); Pytorch (????) to support RWNNs on these setups requires tackling a few rising challenges. First, unlike conventional DNNs, finding an efficient splitting plan of the model is not straightforward. The random connectivity between the nodes in the DNN graph exponentially increases

the mapping's solution space, making our scheduling an NP-Complete problem. Second, the operators' execution order can affect the overall memory footprint for each FPGA, which can become an issue for FPGA setups with limited available memory. This is due to having different execution options for the RWNN graph. Third, even with perfect schedulers, the large difference of computational overhead between the layers can adversely affect the balance of load (and ultimately the performance) between the FPGAs.

To support RWNNs on multi-FPGA setups, we propose a novel scheduler that efficiently splits and maps the layers on the FPGAs and enables throughput maximization on the pipeline. In summary, our thesis makes the following contributions:

(1) Throughput-aware mapping of layers onto the FPGA pipeline. Mapping the layers onto the FPGAs is a complex topological ordering problem, which is NP-Complete. We provide a heuristic scheduling algorithm which offers a near-optimal solution efficiently. We further evaluate various optimization strategies in our algorithm to assess their performance and effectiveness.

(2) Memory-aware scheduling of sub-graph operations on an FPGA. The execution order of the operations on an FPGA affects the maximum memory consumption (memory footprint) on the device. A proper ordering can relax memory footprint to a lower value. In this work, we extend the scheduler to optimize the memory footprint on each FPGA through an appropriate operation execution order.

(3) FPGA design modification to support RWNNs. RWNNs resemble a different regiment of data communication and operation types, which current FPGAs do not support. As a result, the available designs need to be extended to be aware of the random data delivery and the new set of operations used in RWNNs (such as depth-wise separable convolutions).

(4) Operation division. The heterogeneity in the operations' computational complexity can prevent the scheduler from getting the perfect balance of the FPGA pipeline load. This problem can be alleviated by breaking the large operations into multiple small operations and re-executing the scheduler.

We believe that these are significant problems and that solving them would support the effective use of heterogeneous architecture to address the above edge-computing issues and prepare the next generation of the edge devices for the emerging IoT applications.

1.5 Problem Statement

We propose *EdgeFPGA*, an FPGA-Based Edge-computing acceleration solution, that addresses the previous challenges, in the following aspect:

1. Are FPGAs generally suitable for edge computing?
2. How FPGAs are compared to other processors, while accelerating algorithms with different characteristics?
3. How FPGAs are suitable as an accelerator for the emerging deep learning applications? Can they scale to increase the throughput of deep neural network executions?
4. Are the multi-FPGA systems suitable for the emerging generation of neural network architectures, specifically randomly-wired neural networks?

1.5.1 Contributions

The first contribution is the comprehensive study of the accelerators suitability for the edge computing. First, it studies the sensitivity of processing throughput on

both FPGA and GPU, with respect to the workload size of the application. Second, it investigates the adaptiveness of both accelerators to algorithm concurrency and dependency degrees, which are important to edge workloads. Third, it studies the energy-efficiency of the accelerators while running algorithms with different characteristics.

The second contribution is the classification of the common loop patterns and the comparison of their respective performance on different accelerators. First, it applies the identification and classification of common loop patterns in computationally intensive applications. Second, it performs optimization on these loops patterns on the OpenCL-enabled FPGAs and GPGPUs. Third, it evaluates the acceleration potential of these loop patterns on two different accelerators, concerning key configuration parameters, such as computational intensity, dependency and concurrency degrees, and input data size.

The third contribution is the development of a new class of distributed heterogeneous system (based on the CPU and the FPGA) for the streaming AI applications. In this new system, an AI model will be distributed among the available CPUs and FPGAs. Each accelerator handles a specific part of the model. The accelerators are connected in a pipelined fashion, where the data is received in an input channel, traverses through the system, and the output is being streamed to the user through the final accelerator. Unlike conventional distributed systems, this new architecture can provide high-bandwidth and low-latency services, with much higher energy-efficiency.

The Final contribution is the design and development of a scheduler for mapping randomly-wired neural networks on multi-FPGA pipelines. The scheduler guarantees maximum throughput and minimal memory consumption by proposing three main techniques: (1) throughput-aware mapping of layers onto the FPGA pipeline, which is a heuristic algorithm to balance the load amongst the FPGAs in the pipeline, (2) memory-aware scheduling of sub-graph operations on an FPGA, by recognizing a proper ordering of operations execution, which can relax the memory footprint to a lower value, and (3) operation division, which breaks large operations into smaller operations, and enables scheduler to achieve perfect balance on the FPGAs.

1.5.2 Outline

The rest of the dissertation is organized as follow: Chapter 2 describes the background; Chapter 3 presents the study on heterogeneous processors in the edge; Chapter 4 presents the loop acceleration benchmark in heterogeneous systems; Chapter 5 presents a multi-FPGA acceleration framework for the AI application; Chapter 6 presents our novel scheduler for mapping randomly-wired neural networks on multi-FPGA platforms.

Chapter 2

BACKGROUND

2.1 Edge Computing Paradigms

Edge computing is a novel paradigm which brings computation and data storage closer to the devices where it's being gathered, rather than sending data to a remote location (generally cloud) that can be thousands of miles away (Figure 2.1). This paradigm helps latency-sensitive application to achieve real-time performance. In addition, it can reduces costs by having the processing done locally (or near-locally), reducing the amount of data that needs to be processed in a centralized or cloud-based location Abbas *et al.* (2017); Premsankar *et al.* (2018).

Exponential growth in IoT devices motivated the development of the edge computing paradigm. These devices are usually connected to the internet for either receiving information from the cloud or delivering data back to the cloud. Most IoT devices are

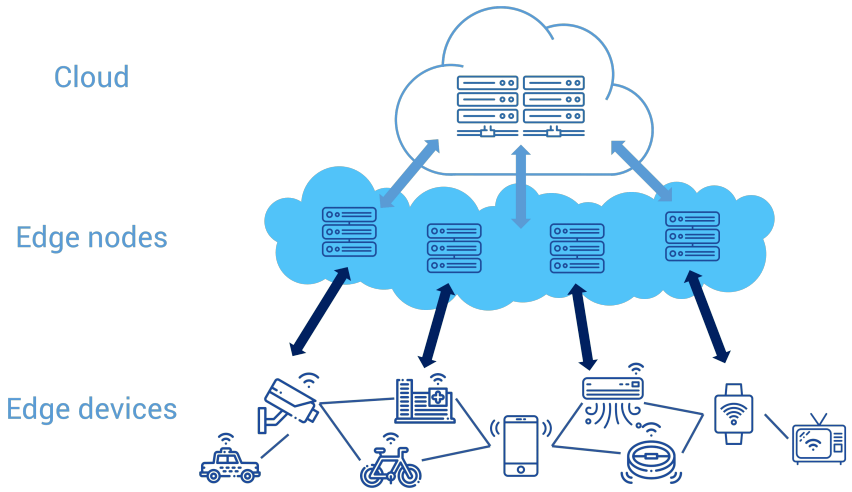


Figure 2.1: Edge-computing Architecture. Source: *Alibaba Cloud. 2020. What Is Edge Computing?*

generating massive amount of data. Examples are: devices that monitor manufacturing equipment on a factory floor, or a CCTV video camera that sends live footage from a remote geo-location. These scenarios can easily overwhelm the capacity of the cloud and internet, which requires computing paradigm rethinking. Edge-computing architecture can help solve the aforementioned problem by acting as the local processing gateway for many use cases. For example, a local edge cloudlet can process incoming video from street camera to perform traffic detection, and then send only the relevant data back through the cloud, reducing bandwidth.

Edge-computing introduces several benefits, which cannot be given by the available cloud-computing paradigm. Here we provide an iteration over these benefits:

Time-Sensitivity. Edge-computing workloads are typically serving a heterogeneous set of applications that demand a tightly-bounded response time Chiang and Zhang (2016). One main example of these applications is IoT devices. Typical IoT devices and sensors, such as cameras, robotic arms, temperature sensors, etc. are constantly generating data and sending requests (along with the input data) to the service providers. These providers are running services that receive these requests, process them alongside the input data, and send the result back to the device. Most of these devices are required to react to an environmental or a user input and make fast and predictable actions, respectively. Table 2.1 represents the average acceptable delay for different sets of applications. Conclusively, the target services are required to: (1) provide a real-time (or near real-time) responses, and (2) send the result back in a timely-predictive manner. Need to mention that typical IoT and user devices lack the proper resource requirements (such as CPU and I/O) to process the requests locally.

The time-sensitivity characteristic of the emerging IoT devices and sensors sets them apart from the traditional cloud-computing applications. The traditional cloud services are mainly designed for: (1) fulfilling time-insensitive applications in a cen-

Service Type	Acceptable Delay
Online Games	< 1000 *ms*
Omnipresent	1000 *ms*
Third person avatar	500 *ms*
First person avatar	100 *ms*
Audio Services	< 450 *ms*
Voice over IP	200 *ms*
Video Services	< 150 *ms*
Video over IP	70 *ms*
Data	< 400 *ms*
Medical Data Transfer	100 - 400 *ms*
Tele-surgery	300 *ms*
Electrocardiogram	1000 *ms*
Non Real-time Services	Few Seconds

Table 2.1: Acceptable Delays for Different Services. Claypool and Claypool (2006); Dusi *et al.* (2012); Skorin-Kapov and Matijasevic (2010)

tralized environment; (2) serving interactive requests from end-users; and (3) processing batches of data, coming from a single source. In contrast, edge-computing services have to service time-sensitive applications from IoT and also end-user devices. We conclude that *Time-Sensitivity* is one of the essential characteristics of edge-computing workloads that have limited existence in the previous generation of workloads.

Location-Awareness: The distribution of the IoT and end-user devices has introduced heterogeneity as one of the main characteristics of the emerging edge-computing

applications. Typical IoT workloads rely on the end-device requests, alongside the historical/geographical relative data to prepare the most proper response **?**. In other words, the same IoT workload that follows a deterministic algorithm to serve the request may rely on dynamic input data to provide the correct answer. For example, a traffic monitoring system may follow the same abstraction to detect a general version of an incident of jay-walking, but it requires the sample data from the local cross-section to provide the highest possible accurate response. Such characteristic makes the edge-computing workloads to operate in a federated manner, as opposed to the traditional centralized computing model.

The sensitivity of the edge-computing workloads to the proximal historical and geographical data makes them different from the legacy cloud computing workloads. Traditional cloud workloads have access to all the data, which makes it hard to adapt to a small group of requests. In contrast, the edge-computing paradigm helps serving requests, for their specific related spatial and temporal information. One may argue that the traditional cloud solution can provide various customized services, based on the sets of specific historical/geographical data and the end-users. Unfortunately, the growth rate of these services is going beyond the capability of the cloud, which is hard to maintain. Also, it cannot fulfill the emerging security requirements of the edge-computing applications. We can conclude that *Location-Awareness* is one of the unique characteristics of the emerging edge-computing workloads.

Security: Achieving proper security for the emerging edge-computing workloads is a challenging and burdensome task. The main differences between the traditional cloud-user workload security model and the edge-computing security model can be iterated as below Hsu *et al.* (2018):

Protocol	Description
Message Queue Telemetry Transport (MQTT)	Lightweight protocol for sending simple data flows from sensors to applications and middleware
Data Distribution Service (DDS)	An IoT standard for real-time scalable and high-performance machine-to-machine communication
Advanced Message Queuing Protocol (AMQP)	An application layer protocol for message-oriented middleware environment
Bluetooth	A short range communication technology integrated into most smartphones and mobile devices
ZigBee	A low-power, low data-rate wireless network used mostly in industrial settings, (6) WiFi, which is the technology for radio wireless networks of devices
WiFi	The technology for radio wireless networks of devices
Cellular	The basis of mobile phone networks, but is also suitable for IoT apps
Long Range Wide Area Network (LoRaWAN)	A protocol for wide area networks

Table 2.2: Most Popular Internet of Things (IoT) Protocols, Standards and Communication Technologies. Hunkeler *et al.* (2008); Beckmann and Dedi (2015); Vinoski (2006); Haartsen (2003); Farahani (2011); Adelantado *et al.* (2017)

- **Heterogeneous Security Protocols:** Different IoT workloads are utilizing different light-weight communication protocols to transmit/receive data between the IoT endpoints, the service-provider cloudlets, and the centralized cloud. Unlike the traditional cloud-user paradigm, where the communication is usually based on a heavy multi-layer standard protocol, the IoT devices are relying on light-weight and customized communication stack. Examples of these protocols are demonstrated in table 2.2. Lack of homogeneity in the edge-computing workloads' communications makes it hard to guarantee a customized and fine-grained secure access control. This specific challenge requires rethinking the traditional security management and enforcement for the emerging edge-computing workloads.

- **Limited Computing Resources:** Requests for the edge-computing workloads are mainly generated by the massive number of IoT devices. These devices are usually sensors and small-sized cameras, with limited computational and storage capabilities. Traditional security protocols require expensive cryptography processing, which is out of the capabilities of such devices. As a result, edge-computing workloads are not able to adapt to traditional security mechanisms.

Hsu et al. Hsu *et al.* (2018) have proposed a unique edge-computing solution to address the upcoming challenges. In this framework, routers, base stations, and other near-edge boxes acting in this new security role would handle the computing that the IoT devices can't (due to size, power limitations, and so on). Researchers say this will not only be more secure, but it will also simplify the management of keys. Cryptographic key disclosure risk increases as more keys, or passwords, need to be implemented by applications. The solution would also be more scalable. Based on the above challenges, edge-computing workloads request for new security enforcement.

2.2 Edge Computing Platforms

Computing the IoT requests is one main aspect of the edge platforms, which should be provisioned inside the edge cloudlet servers. Existing edge servers on the market are simply a miniature version of cloud servers which are primarily structured based on CPUs with tightly coupled co-processors (e.g., GPUs). However, CPUs and GPUs are optimized towards batch processing types of workloads, and cannot fulfill latency-sensitive and streaming IoT requests. Therefore, the next generation of the edge servers should come with a complimentary processing element, which directly aims toward the edge-computing workloads. These processing elements should embody three specific features. First, they need to be able to service streaming workloads in a real-time (or near real-time) fashion. While CPUs and GPUs can generally provide reasonable performance, they cannot fulfill workloads with a certain type of latency boundaries. As a result, edge servers should be equipped with processors that can provide fast responses for individual requests. Second, they need to support a wide variety of applications with different characteristics. Edge-computing workloads cover a wide variety of services, with various computing patterns. These patterns are reflected as the degree and the type of flow and the dependency in the algorithm. The upcoming edge server processors should be able to provide reasonable coverage for a wide array of edge applications. Third, they need to be suitable for both single and batch of input requests. IoT requests can come in different quantity and shapes. An appropriate edge-computing platform should efficiently support all of these requests.

Energy-efficiency is an important part of the edge-computing platforms. These platforms are going to be deployed as cloudlets, with variable scales in a geo-distributed fashion. Unlike centralized cloud data centers, many of the cloudlets may face limited energy availability in various geographical locations. Besides, cooling can be

another major problem for servers with a considerable amount of heat generation. To guarantee maximum availability, edge-servers need to be redesigned to operate in a low-power mode, while maintaining good computational efficiency. Available processors in the current edge servers (CPUs and GPUs) are power hungry and have limited energy-efficiency and are creating enormous difficulties for deploying them in energy- and thermal-constrained scenarios. As a result, power-consuming components of these servers need to be replaced with other identical counterparts that consume much less energy and provide better or equal energy-efficiency.

Edge-computing platforms (more specifically, edge-computing servers) need to be flexible enough to support various I/O channels. In an edge-computing setup, the majority of the requests are coming from the IoT devices. Each type of these IoT sensors is equipped with a different type of I/O transmitter/receiver. Unlike the centralized cloud model, where the users and cloud servers are all equipped with standard communication mediums, in edge-computing paradigm every device is using different communication hardware. Enabling communications between two different hardware mediums requires the addition of hardware and software translators into the edge-servers and IoT stacks. Unfortunately, the IoT devices cannot expand beyond their dedicated hardware resources. As a result, the edge-servers need to compensate for this problem, by adding support for various I/O communication protocols, either by using the software or the hardware.

2.3 Hardware Acceleration and FPGAs

Hardware acceleration is a crucial enabler of the High-Performance Computing (HPC) applications. Due to the computational intensity of HPC workloads, CPUs cannot deliver a reasonable performance for latency-critical applications. This problem leads to the utilization of hardware accelerators, such as GPUs, FPGAs, and

TPUs Jouppi *et al.* (2017). Unlike CPUs, hardware accelerators can exploit their massive parallelism to split major functions into thousands of parallel operations, and ultimately reduce the overall computation time. GPUs have been extensively studied and utilized for the acceleration of HPC workloads. While GPUs are highly effective in handling applications with high level of concurrency and regular memory access patterns, they come short for applications with a high degree of dependency, and/or a high number of conditional branches. Examples of these applications include graph processing Cong *et al.* (2018b), sorting Koch and Torresen (2011), small signal processing problems Duan *et al.* (2011), and sparse linear algebra Zhang *et al.* (2009). Widely-used deep learning frameworks, such as TensorFlow Abadi *et al.* (2016) and Caffe Jia *et al.* (2014), rely on GPUs to deliver acceptable performance for both the training and the inference. Recently, FPGAs have captured the right amount of attention due to their flexibility and reconfigurability. FPGAs are proven to be able to provide much lower latency, compared to CPU and GPU, for applications with latency-critical conditions Zhang *et al.* (2016, 2018). Different from widely-adopted GPUs and CPUs, FPGAs can accelerate almost all types of algorithms (irrespective to their computational pattern), due to their reconfigurability. Also, they can provide a much better energy-efficiency, compared to CPUs and GPUs, which is crucial for energy-restricted environments, such as edge computing Biookaghazadeh *et al.* (2018); Zhang *et al.* (2016).

Recent advancements in high-level languages have made it easy to program and use accelerators, specially FPGAs, for various applications. For example, developers can use C or C++ to describe their algorithm and compile and deploy it on a target accelerator. Hardware accelerator vendors have integrated OpenCL, a heterogeneous parallel programming language, with their platforms. OpenCL has several benefits for software developers and systems designers. It provides ease of development by

keeping a higher abstraction, at the cost of an acceptable performance loss. Also, it enables software engineers to take advantage of the ultimate performance and the energy-efficiency of an available platform. Using OpenCL, developers can describe their algorithm in standard representation, and target all available accelerators, such as GPUs, CPUs, and DSPs. To port OpenCL across different platforms, a developer needs only to make minor modifications to utilize the unique features of the target platform fully.

2.3.1 Parallelism

Algorithms can be parallelized either *temporally* or *spatially*.

Spatial Parallelism. In *spatial* parallelism Freitas and Lavington (2000), processing elements (PEs) execute the same task (SIMD) or multiple different tasks (MIMD), simultaneously. Both GPU and FPGA are able to exploit spatial parallelism in algorithms. The amount of data dependency between the iterations of the loops in the algorithm can decide the level of achievable spatial parallelism on the target architecture. In another word, having less data dependency increases the opportunity of speedup on parallel architectures, such as GPUs and FPGAs. In general, GPUs are better at exploiting spatial parallelism, because FPGAs cannot adopt as many compute cores as GPUs, and FPGAs also tend to operate at a lower clock frequency, up to 2-5 times slower than GPUs.

Temporal Parallelism. In *temporal* parallelism Freitas and Lavington (2000), processing tasks that have a dependency on each other are mapped onto different PEs and execute in parallel in a pipeline fashion. Data processing has multiple stages, and each stage is being handled by one PE. In this multi-stage pipeline, as data is processed by the element PE_i, it is sent to the next element PE_{i+1} and element PE_i moves on to handle new data coming from the previous stage. In the cases where a

single task cannot fully occupy the available PEs, multiple tasks can be interleaved and mapped onto the PEs to increase the *temporal* parallelism.

Among general purpose accelerators, FPGAs are exclusively able to exploit coarse-grained temporal parallelism in the algorithms, due to their reconfigurability. SIMD platforms like GPU can perform at most one instruction at a time on each available core, whereas FPGA can execute hundreds of operations on all available stages in the pipeline. Need to mention, while GPUs can launch multiple kernel streams in a pipeline fashion, they cannot achieve the fine-grained pipeline parallelism. One can mimic pipeline parallelism by launching consecutive kernels (e.g., CUDA stream kernels). Still, the data between different stages should be stored and delivered to the main memory, an expensive operation. On the contrary, FPGAs utilize connected registers between PEs to transfer the data.

Figure 2.2 depicts both parallelism dimensions. Each circle represents an individual iteration in a set of nested loop blocks. The (i, j) pair in each circle represents the ith iteration in the first dimension and the jth iteration in the second dimension. The arrow represents the dependency of one iteration on another, e.g., (1,2) depends on (1,1). Each iteration usually involves separate calculation for a specific indexed item or accumulation on a shared value among iterations of a loop block. The dashed box contains iterations with zero dependency, which can be easily parallelized spatially. On the other hand, the dotted box contains iterations with data dependency, which cannot be parallelizaed spatially but may have the potential to be parallelized temporally. We use the above format throughout the paper to represents the dependency flow.

In summary, GPUs excel at exploiting spatial parallelism but cannot utilize temporal parallelism, whereas FPGAs can take good advantage of both types of parallelism. However, despite this general understanding of GPU's and FPGA's different

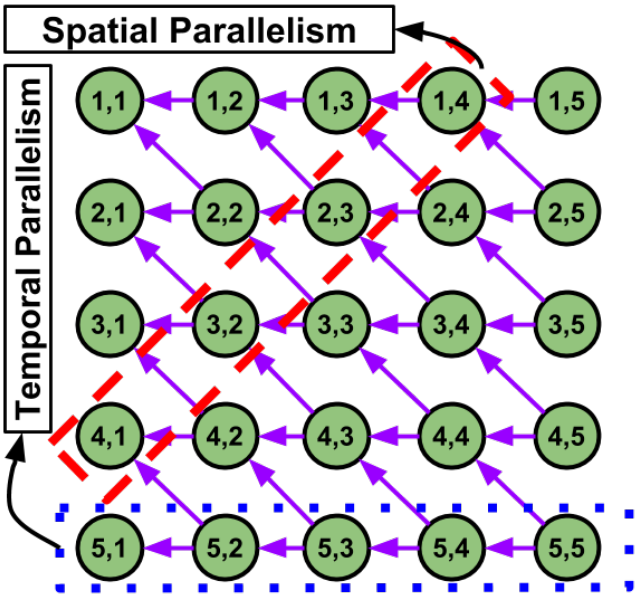


Figure 2.2: Spatial and Temporal Parallelism in Multiple Iteration Dimensions.

strengths, it is still difficult to understand which accelerator works the best for which algorithm. Every single application consists of different types and degrees of conditional and data dependencies. Developers usually need to implement the code for different accelerators and then apply several different transformations on the algorithm to assess the acceleration potentials on different devices. Understanding the relationship between common micro-level patterns such as loop patterns and their potential acceleration can reduce the effort of choosing the right device. These are the motivations for our study on loop acceleration using GPUs and FPGAs, which, to the best of our knowledge, is the first.

2.3.2 Automatic Loop Optimization

Algorithms are composed of one or many loops, either nested or flattened. The acceleration of algorithms is the process of acceleration of the loops, using parallelization and pipelining methods.

Automatic loop optimization (generally parallelization) dates back to an article by

Lamport Lamport (1974) which discusses parallel execution of *do loops*. Later, several other researchers continued the effort and developed the groundbreaking approach of using linear algebraic methods to analyze, transform, and parallelize loops, namely polyhedral compilation Cousot and Halbwachs (1978); Schreiber *et al.* (1990); Ancourt and Irigoin (1991); Loechner (1999); Bastoul (2004).

Polyhedral compilation is used in a wide range of applications, including automatic parallelization, SIMDization, code generation for hardware accelerators, and memory and cache consumption optimization. It models nested loops and arrays into an algebraic format while presenting specific constraints, such as dependencies. Further, it uses particular types of algebraic transformation that guarantee the loop's semantic and correctness and generates a new model, typically optimized toward a specific cost model. Finally, the model is translated back into an execution code that can run on hardware.

The polyhedral compilation has limitations. First, it does not provide cross-accelerator comparisons. Such limitation prevents developers from understanding the correlation between the loop patterns and the speedup capabilities of accelerators. Second, polyhedral compilers, such as Polly Grosser *et al.* (2011, 2012), Graphite Trifunovic *et al.* (2010), and a more recent compiler called Tiramisu Baghdadi *et al.* (2019) can only optimize specific routines in domain-specific applications, such as dense linear algebra, tensor operations, and stencil computations. Also, they are only able to provide roughly 10% performance improvement Simbürger *et al.* (2013). Finally, the polyhedral compilation is not well-studied on GPUs and FPGAs compared to CPUs Konstantinidis *et al.* (2013); Juega *et al.* (2014); Wang *et al.* (2021), making it less effective for accelerators.

In summary, polyhedral compilation lacks the ability to demonstrate the effectiveness of different hardware accelerators while considering an algorithm or an applica-

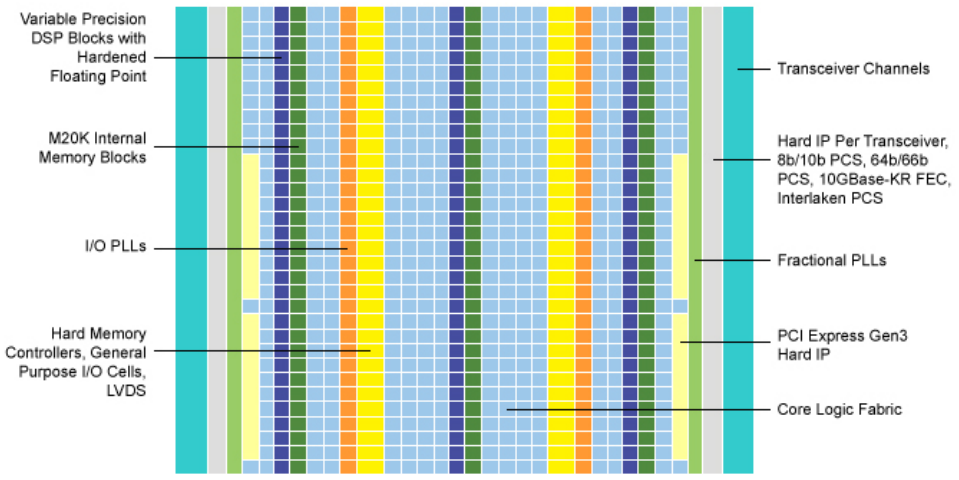


Figure 2.3: Intel Arria 10 FPGA Internal Architecture. Source: *Intel Inc ©*.

tion. Loopy aims to unlock insights into accelerating typical loop patterns that can be generally found in many applications with important accelerators such as GPUs and FPGAs.

2.3.3 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are a farm of logic, computation, and storage resources that can be configured dynamically (Figure 2.3 depicts a sample Intel Arria 10 FPGA architecture). FPGAs can be reconfigured to execute an algorithm in a dedicated form. They have successfully been used in many application domains. Despite their impressive acceleration power, programming and optimization difficulties have been serious obstacles to the wider adoption of FPGAs. Recent advancements in supporting high-level synthesis (HLS) have made it possible to program FPGAs using high-level languages, especially OpenCL Munshi (2009), which has made FPGAs much easier to use and much more accessible to applications. Even though an HLS-based program may not perform as well as a carefully hand-crafted HDL program, the productivity enabled by HLS is often far more important.

Application	Device	Latency (seconds)
Fractal Video Compression Chen and Singh (2013)	CPU	0.217
	GPU	0.018
	FPGA	0.013
Real-time Stereo Vision Kalarot and Morris (2010)	CPU	N/A
	GPU	0.05
	FPGA	0.033
Convolutional Neural Networks Zhang *et al.* (2018)	CPU	0.73
	GPU	0.023
	FPGA	0.025

Table 2.3: Latency comparison between GPU, CPU and FPGA

The unique features of the FPGAs make them a great candidate for future edge-computing platforms. Below we iterate on how FPGAs can fulfill edge-computing workload demands:

- **Accelerating time-sensitive applications:** FPGAs can easily be reconfigured to accelerate a specific algorithm in a dedicated fashion. This unique feature can serve a time-sensitive application from two different perspectives. First, the FPGA can specifically reflect the execution path of the algorithm on the chip, which helps to avoid all the overheads that exist in the general-purpose accelerators and provide much lower latency. Second, It can guarantee a tight latency boundary for the target IoTs or the user devices since the number of

cycles and the length of each cycle to finish the whole execution on the board is known beforehand. Table 2.3 represents the capability of various accelerators in executing sample applications. FPGAs can deliver better latency for handling a single piece of data, compared to CPUs and GPUs.

- **Accelerating streaming input:** Unlike GPUs and CPUs that are optimized for batch processing of the data from the memory, FPGAs are inherently efficient for accelerating streaming applications. A pipelined streaming architecture with a data flow control can be easily built on an FPGA to process streams of data and commands from I/O channels and generate output results at a constant throughput with reduced latency.

- **Adaptiveness to algorithm characteristics:** Edge-computing platforms are required to service a wide variety of applications in the cloudlets. FPGAs can adapt to any algorithm characteristics due to their hardware flexibility. Different from CPUs and GPUs that can mostly exploit *spatial* parallelism, FPGAs can exploit both *spatial* and *temporal* parallelism at a finer granularity in a larger scale. FPGAs can construct both types of parallelism using their abundant computing resources and pipeline registers. Biookaghazadeh et al. Biookaghazadeh *et al.* (2018) have demonstrated the feasibility of the FPGAs in handling loops with variable data dependency. Higher data dependency among the iterations leads to less opportunity for spatial parallelism and higher opportunity for temporal parallelism. In their experiments, FPGA can adapt and exploit the temporal parallelism.

- **Energy-Efficiency:** FPGAs consume significantly lower power compared to CPUs and GPUs for delivering a comparable throughput, allowing for improved thermal stability and reduced cooling cost. Table 2.4 represents the total power

Application	Device	Power (watts)
Sliding-Window Application Fowers *et al.* (2012)	CPU	130
	GPU	274.5
	FPGA	20
Fractal Video Compression Chen and Singh (2013)	CPU	130
	GPU	215
	FPGA	25
Dense Linear Algebra Zohouri *et al.* (2016)	CPU	78.64
	GPU	184.41
	FPGA	29.48
Convolutional Neural Networks Zhang *et al.* (2016)	CPU	87.3
	GPU	328.3
	FPGA	19.1

Table 2.4: Power consumption comparison between GPU, CPU and FPGA

consumption of common accelerators while executing widely-used algorithms. Based on these results, FPGAs consume around 16 and 3.59 times less power, compared to the GPU and the CPU. This merit is critically needed for edge servers, considering their limited form factors.

Despite all the above benefits, FPGAs lack certain benefits, compared to GPUs and CPUs. Below we iterate on how FPGAs come short comparing to GPUs and

CPUs, specially on edge workloads:

- **Low clock frequency:** The operation clock frequency of the FPGAs is usually between 200 to 300 MHz. This is much less than the clock frequency of CPUs and GPUs, which can go beyond 2GHz. This massive difference can cause FPGAs to under-perform for certain applications with certain input types and sizes. For example, with the same amount of parallelism on an FPGA and another accelerator, the higher clock frequency can lead to much higher performance. As a result, the FPGA may perform poorly in specific applications, compared to the other accelerators.

- **Limited hardware resources:** FPGAs are equipped with certain types of resources, such as Digital Signal Processors (DSPs), Lookup Tables (LUTs), Flip-Flops (FFs), etc. The total number of these resources are limited by the physical area of the FPGA chip. Compared to the ASIC accelerators (CPU and GPU), FPGAs can adopt a much smaller number of processing elements. As a result, they can deliver a limited amount of parallelism in the space domain (spatial parallelism). This limitation makes FPGAs not suitable for applications with a high degree of spatial parallelism.

- **Difficult programmability:** Originally, FPGAs can be programmed using hardware description languages (HDLs), such as VHDL and Verilog. Recent advancements have improved the programmability of these devices, by enabling developers to use higher-level programming languages, such as C/C++, to describe and compile their algorithms on the target FPGAs. Unfortunately, even with the availability of these languages, hardware developers need to spend a considerable amount of time to fully customize the design for the FPGA, to get the highest possible performance. Hence, FPGAs is only accessible by a limited

group of hardware developers, but not the general software community. On the other hand, GPUs and CPUs are supported by highly efficient compilers and a large number of libraries, which makes them accessible to a diverse group of users.

Recent observations of the workload in various domains, such as IoT and cloud, have detected a common set of operations and utilities (we call them idioms) that are frequently used in various applications. As an example, Deep Neural Networks (DNNs) are one of the emerging applications on the edge. DNNs are vastly being used in image and video processing for object and motion detection, and many other useful applications. The main building block of these DNNs are convolutions and matrix multiplications, where the convolutions can also be represented as matrix multiplications. As a result, matrix multiplication can be counted as one of the widely-used idioms. Another example is the serialization and the deserialization of the data over the networks, between the applications and the services. A recent profiling **?** by Google shows a considerable share of the above functionalities in the Google data center. Acceleration of these idioms can improve the performance of a diverse set of applications. Need to mention that the same trend has been observed in mobile computing applications, which has led to the development of the SoC architectures.

With the ever-growing adoption of the FPGAs, software, and hardware developers are required to repetitively develop and map the above idioms, which can reduce the overall productivity and will consume a large number of valuable resources on the chip. The new FPGA chip technologies are moving toward the adaptation of coarse-grained resources, such as ARM cores, vector processors, etc. For example, the new FPGA family from Xilinx, which is called ACAP, is geared with various computing components. Based upon the ACAP's description Vissers (2019), it is equipped with three main components, which are the scalar engine, the adaptable engine, and the

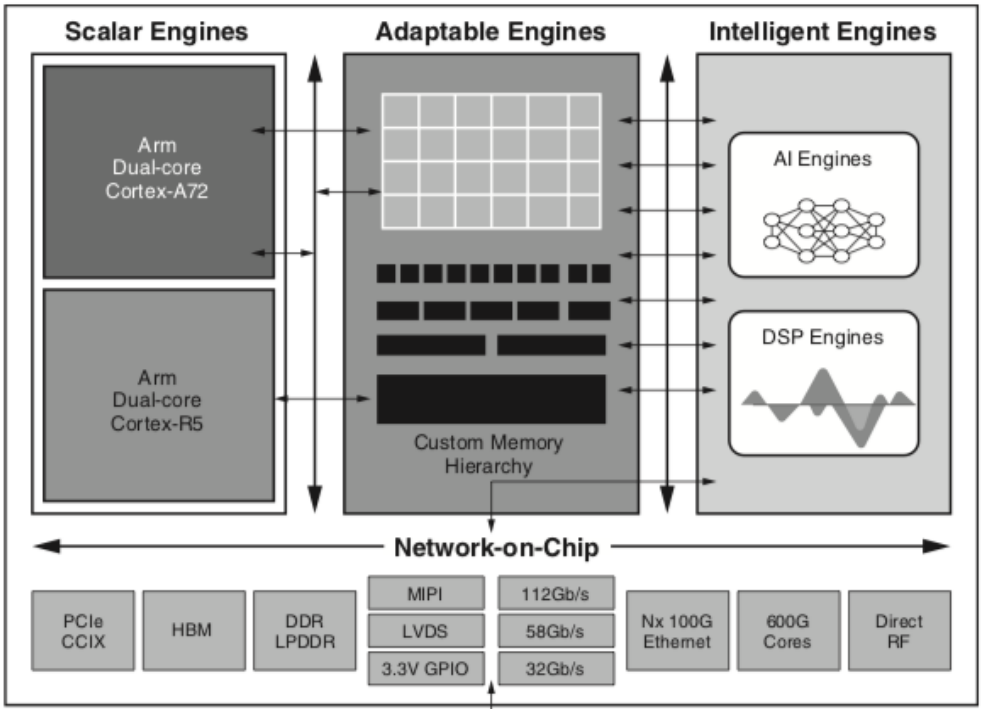


Figure 2.4: ACAP hardware architecture. Source: *Xilinx Inc ©.*

intelligent engine. The scalar engine is built from two dual-core ARM Cortex processors. The adaptable engine is made up of programmable logic and memory cells (an FPGA). Finally, the intelligent engine is an array of innovative VLIW and multiple SIMD processing engines. Figure 2.4 represents the overall architecture of the ACAP chipsets. These resources can be used by the compiler tool-chains and libraries to accelerate the common idioms, and avoid wasting the resources on the chip. Need to mention that the above arguments are also applicable to the available SoC+FPGA accelerators, such as Intel Arria 10 SX family.

The upcoming FPGAs with coarse-grained resources can benefit edge-computing workloads in various ways. First, having resources to accelerate common idioms in the edge application can help to reduce the processing time and better serve the time-sensitive IoTs. It is quite common to have edge cloudlets that are designed

to serve a family of applications in a small area. For example, a cloudlet in a city may provide visual AI services, such as video and image processing. In these cases, having common components, such as encoder/decoder, vector processors, etc. can significantly help the acceleration of these workloads. Second, these resources can lead to better energy-efficiency. The ASIC version of these idioms consumes much smaller hardware area, compared to the FPGA equivalent, which can lead to lower energy consumption and better thermal stability. Third, it makes edge application development easier for developers. Using these components, developers can avoid re-implementing repetitive common idioms, which helps with productivity and time-to-market.

Preliminary results Vissers (2019) from emerging architectures, such as the ACAP, and the other available SoC+FPGA systems reveal promising performance and energy benefits. The ACAP hardware delivers up to 90, 8, and 5 times better performance in various data center and edge applications, such as image recognition, risk analysis, and genomics, compared to the CPU, the GPU, and the FPGA. It is also able to provide up to 5, 100, and 15 times faster run-time performance, compared to the CPU, the GPU, and the FPGA. With respect to the widely-available SoC+FPGA architecture, applications can utilize the available heterogeneity to accelerate different parts of the applications with different computational patterns.

2.4 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are the main building blocks in many AI applications, such as image classification Litjens *et al.* (2017), reinforcement learning Arulkumaran *et al.* (2017), and natural language processing. CNNs are also showing promising results in more complex domains such as video understanding Maturana and Scherer (2015); Hegde *et al.* (2018); Tran *et al.* (2015); Sun *et al.* (2015); Ji *et al.*

(2012). Almost all CNNs are considered as a chain of various operations, such as convolution (2D or 3D), matrix multiplication, pooling, and ReLU. In CNN, the data is processed by one operation, and the result is handed over to the next operation in the chain. In a 2D convolution operation, the input data is just composed of multiple input channels, where each input channel is a two-dimensional structure of numerical values. In a 3D convolution operation, the input data is not only composed of multiple input channels, but each input channel contains data from different instances in a time frame, where the instances should be sequential in that specific time frame. Equation 2.1 and Equation 2.2 describe the 2D and 3D convolutions, where m represents a specific output channel, f represents a specific frame number, w and h represent width and height location respectively, in the output, CH_{in} represents the total number of input channels, n, k, i, and j represent the iterator indexes on the input channels, frames, and width and height locations of the convolution kernel.

$$OUT[m][w][h] = \sum_{n=0}^{CH_{in}} \sum_{i=0}^{K_w} \sum_{j=0}^{K_h} WEIGHT[m][n][i][j] \times IN[n][stride \times w + i][stride \times h + j] \tag{2.1}$$

$$\begin{aligned} &OUT[m][f][w][h] = \\ &\sum_{n=0}^{CH_{in}} \sum_{k=0}^{K_f} \sum_{i=0}^{K_w} \sum_{j=0}^{K_h} WEIGHT[m][n][f][i][j] \times \\ &IN[n][stride \times f + k][stride \times w + i][stride \times h + j] \end{aligned} \tag{2.2}$$

CNNs are highly computationally intensive, due to a large number of mathematical operations (hundreds of thousands and even up to millions) that each layer of the network involves. Amongst all the widely-used operations in the neural networks, convolutions and matrix multiplications (also known as fully-connected or FC) are

Operataion (2D)	Ops	Data
Convolution	99.19%	8.61%
Matrix Multiplication	00.79%	91.38%
Pooling	00.00%	00.00%

Table 2.5: Total Contribution of Major Operations in VGG-16 CNN Model, in Terms of Total Number of Arithmetic Operations and Input/Weight Parameters.

the most significant contributors to the total execution time for one round of inference on a simple neural network. Table 2.5 reports the contribution of three primary operations in the VGG-16 model, in terms of the total number of arithmetic operations (such as multiply-and-accumulation (MAC), min, and max) and the parameter size. *Ops* and *Data* columns represent the total number of arithmetic operations and parameters (weights and inputs) involved in that operation, respectively. The convolution operations (2D) contribute more than 99% of the total arithmetics. The matrix multiplication operations contribute more than 91% of input and weight data access from global memory, which can consume a considerable portion of the total runtime.

Operataion	Ops	Data
Convolution (3D)	99.9%	26.72%
Matrix Multiplication	00.1%	73.28%
Pooling	00.00%	00.00%
ReLU	00.00%	00.00%

Table 2.6: Total contribution of major operations in C3D CNN model, in terms of total number of arithmetic operations and input/weight parameters.

Compared to 2D convolutions, 3D convolutions have higher computational complexity, due to the existence of an extra dimension (usually frame), which enables spatio-temporal feature recognition in continuous video frames. Table 2.6 lists the total contribution of 3D convolutions and other operations in the C3D model. The convolution operations (3D) contribute more than 99% of the total operations. The matrix multiplication contributes to more than 73% of data access.

2.4.1 Winograd Algorithm

Winograd transformation Lavin and Gray (2016) is a proven method to reduce the complexity of multiply-accumulate operation in hardware design. Using this technique for convolutions can ultimately reduce the arithmetic complexity. Shen et al. Shen *et al.* (2018) showed that using the Winograd algorithm can reduce the total number of multiplications by 58%. Also, Winograd becomes more practical for smaller filter sizes, such as 3x3, which is quite common in many neural networks. In our design, we utilize the 2D Winograd algorithm to accelerate both 2-D and 3-D convolutions on the FPGAs. To demonstrate the Winograd algorithm, we will start with an example of a one-dimensional (1D) convolution. In the Winograd algorithm, we denote a 1D convolution as $F(M, R)$, where M and R represent the size of the input and the filter. The typical convolution computation is given by:

$$O_i = \sum_{r=0}^{R-1} W_r I_{i+r} \tag{2.3}$$

where I, O, and W denote the input, output, and filter data. By using Winograd algorithm, the output can alternatively be derived as follows Winograd (1980) (we consider Winograd algorithm for $F(2, 3)$):

$$O = M[(Sx) \cdot (Ww)] \tag{2.4}$$

where M, S, and W are transformation matrices with values of:

$$S = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad , \quad W = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{-1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{-1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \quad , \quad M = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \tag{2.5}$$

The above method can be extended for 2D convolutions, as well. Considering 2D Winograd algorithm $F(m \times m, r \times r)$, it can be calculated using the following equation:

$$O = M[(SxS^T) \cdot (WwW^T)]M^T \tag{2.6}$$

2.4.2 CNN Acceleration on FPGAs

Several related works have studied the acceleration of the 2D Aydonat *et al.* (2017); Zhang *et al.* (2018); Wang *et al.* (2017); Zhang *et al.* (2015); Suda *et al.* (2016); Ma *et al.* (2018) and 3D CNNs Liu *et al.* (2019); Ji *et al.* (2012); Tran *et al.* (2015); Hegde *et al.* (2018) on FPGAs. Other related works Jiang *et al.* (2019); Zhang *et al.* (2016) have studied the feasibility of using multiple FPGAs for increasing the throughput or decreasing the latency of the CNN accelerators. However, these works cannot deliver state-of-the-art performance and are not designed to support different types of convolutions in a single architecture. Also, they are all implemented with low-level hardware languages, which makes them hard for further extensions and improvements. Our design is built on top of the DLA Aydonat *et al.* (2017) architecture. Boutros et al. Boutros *et al.* (2018) made a comparison between widely-known CNN accelerators on FPGA and showed that DLA is the fastest available solution. However, DLA lacks

several important optimizations and critical features. For example, the systolic array needs enhancements for lower resource consumption and higher throughput. Also, weight and input organization can be changed for better memory utilization. From the usability perspective, it works for only the default 2D convolution but cannot support the more complex 3D convolution. Finally, it does not support the multi-FPGA acceleration, which is important for complex CNNs. Our design is built on top of DLA while addressing all the above limitations.

Some other related works Zhang *et al.* (2016); Jiang *et al.* (2019) have studied the multi-FPGA acceleration of neural networks. These works come with several limitations. First, they do not provide a general architecture to accelerate various types of CNNs. For example, they are only able to accelerate either two-dimensional (2D) or 3D convolutions, but not both. Second, they do not optimally exploit the FPGA acceleration resources, which leads to sub-optimal performance, compared to the maximum theoretical performance of an FPGA. Third, they are designed and developed, using low-level hardware programming languages (Jiang et al. Jiang *et al.* (2019) used Xilinx HLS), which makes it difficult to extend and support by the widely-used deep learning frameworks, such as Tensorflow Abadi *et al.* (2016) and Caffe Jia *et al.* (2014).

2.5 Randomly-Wired Neural Networks

Automatic machine learning model discovery and development (AutoML) Feurer *et al.* (2015); Dean (2017); He *et al.* (2018); Yazdanbakhsh *et al.* (2018); Wang *et al.* (2019a); Laredo *et al.* (2019) helped engineers and scientists to explore and build models with much higher accuracies, compared to the conventional hand-tuned models. Examples are *Neural Architecture Search (NAS)* Zoph and Le (2016); Zoph *et al.* (2018); Liu *et al.* (2018a); Cai *et al.* (2018); Real *et al.* (2019); Cheng *et al.* (2019)

and *Random Network Generators* Xie *et al.* (2019); Wortsman *et al.* (2019), which resemble random connections between the operations and are known as *Randomly-Wired Neural Networks (RWNNs)*. Both these techniques are focused on automatic generation of neural network architectures. Figure 1.1 depicts examples of RWNNs (*SwiftNet* and *RandWire*).

RWNNs have been able to outperform conventional neural networks, which are manually designed and tuned. These networks can provide an equal or better performance while using less number of operations and parameters. Figure 2.5 presents an overall comparison between networks generated by automatic methods, such as NAS and Random Network Generators, and other widely-used neural networks (such as ResNet, Inception, MobileNet Sandler *et al.* (2018), etc.). Comparisons are based on the ability of the network to classify the *Imagenet* data successfully.

Most RWNNs make extensive use of depth-wise separable convolutions Sandler *et al.* (2018) which are computationally cheaper and consume less storage than conventional convolutions. Figure 2.6 presents both (a) conventional and (b) depth-wise convolutions. In normal convolution, both the filter and the input has the same number of channels. The filter moves through the input's width and height dimensions and generates a 2-D feature map for a specific output channel. Differently, in depth-wise convolution, filters have a depth of one, and they only convolve with one channel of the input. As a result, the total number of weight features and the input channels are equal.

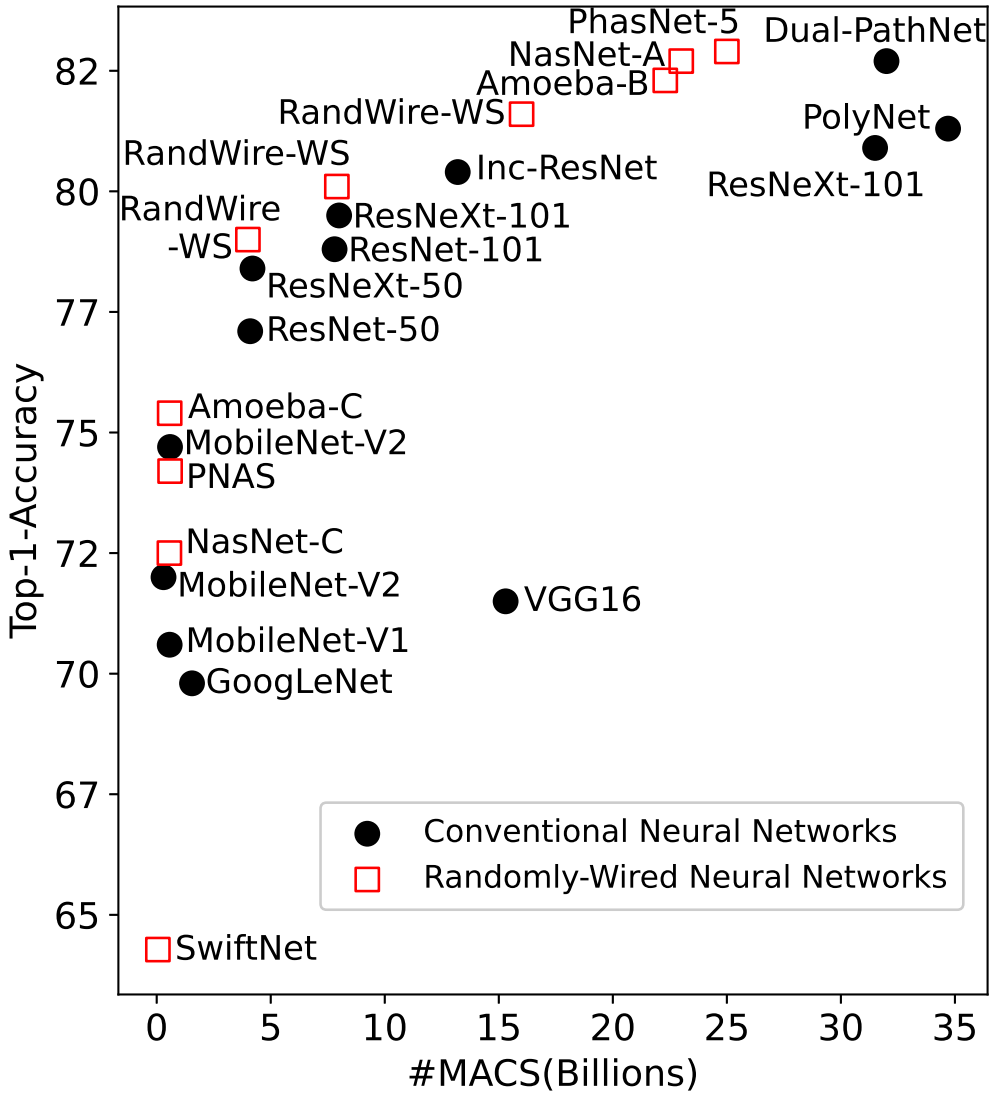


Figure 2.5: The Figure Shows the Performance Comparison Between Conventional Neural Networks and RWNNs. Accuracies are Measured by Evaluating the Models with ImageNet Data. The X-Axis Presents the Total Number of Multiply-accumulate (MAC) Operations of the Model, and the Y-Axis Shows the Classification Accuracy.

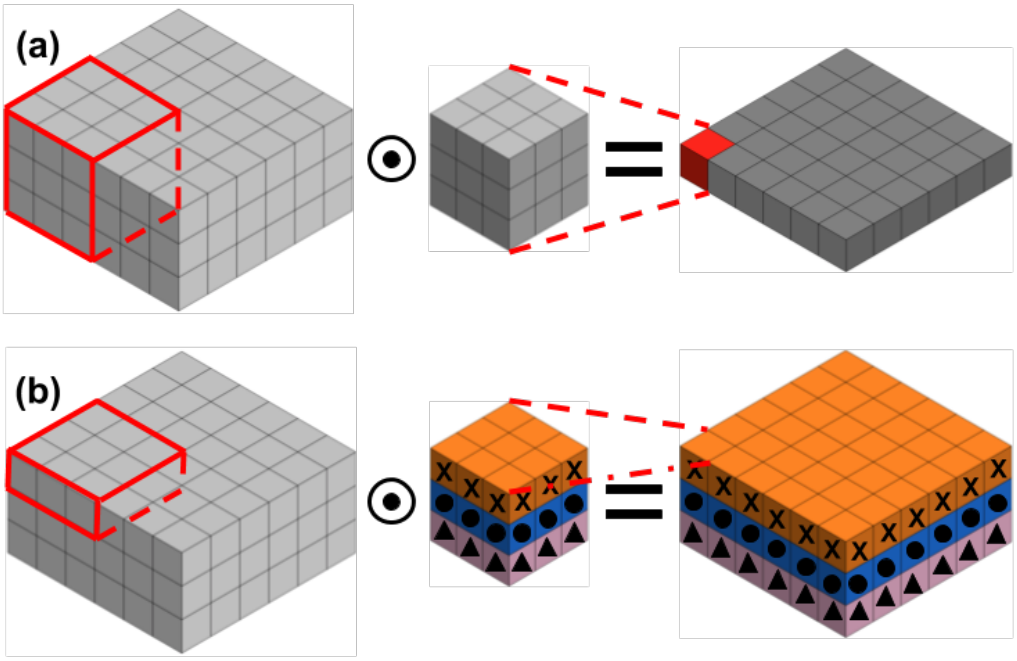


Figure 2.6: Depiction of the (a) Conventional Convolution, and the (b) Depth-wise Convolution.

A simple depth-wise convolution produces the same number of channels as the input data, while the number of channels can be a different number. To enable this feature, a second point-wise convolution is applied. This operation uses a 1×1 kernel, with a depth equal to the number of input channels. The convolution of the intermediate features with this kernel introduces one output channel. The number of kernels is equal to the number of output channels. In summary, the combination of the two operations mentioned above builds the depth-wise separable convolution.

2.5.1 Deep Learning Compilers

The rapid growth of neural network usage in various domains motivated the development of specific compiler frameworks Abadi *et al.* (2016); Paszke *et al.* (2019); Chen *et al.* (2018) for deep learning applications. These frameworks mainly focus

on fine-grained scheduling of (sub)operations on a single general-purpose accelerator (e.g., CPU and GPU) and do not support pipeline accelerators, such as FPGAs. Other available compilers for FPGAs Sharma *et al.* (2016); Biookaghazadeh *et al.* (2020); Abdelfattah *et al.* (2018) either do not support multi-FPGA configuration or are not friendly with RWNNs, due to the complexity of the network architecture. In summary, these frameworks are designed with the assumption of the conventional linearly-oriented neural network models and do not guarantee efficient execution of the emerging RWNNs Zoph and Le (2016); Cheng *et al.* (2019); Wortsman *et al.* (2019); Xie *et al.* (2019).

2.5.2 Scheduling RWNNs

Scheduling RWNNs is not supported by the available deep learning frameworks. More specifically, scheduling RWNNs onto multi-FPGA platforms boils down into a *graph partitioning (GP)* problem, which can be widely found in various domains, such as parallel processing Boman *et al.* (2013); Buluc and Madduri (2013); Sali-hoglu and Widom (2013), road networks Kieritz *et al.* (2010); Maue *et al.* (2010); Luxen and Schieferdecker (2012), image processing Peng *et al.* (2013); Camilus and Govindan (2012); Grady and Schwartz (2006), and VLSI physical design Kahng *et al.* (2011); Cong and Shinnerl (2013). In this specific graph partitioning problem, we deal with an acyclic graph with weighted nodes but not necessarily weighted edges (due to negligible data transfer overhead). Several related works have studied the graph partitioning problem with various constraints. In general, the partitioning problem is NP-complete Hyafil and Rivest (1973); Garey *et al.* (1974), which has urged researchers to explore effective heuristics.

There are different categories of related works studying the graph partitioning problem. Exact algorithms Hager *et al.* (2013); Sensen (2001) can derive a solution

for the whole graph, usually relying on branch-and-bound framework Land and Doig (2010). These algorithms are suitable for small problem sizes due to their large running times. Graph growing Karypis and Kumar (1998a); Duff (1984) can obtain a bisection of the graph, using a breadth-first search (BFS) from a random node v. Flow algorithms Bui *et al.* (1987); Lang and Rao (2004) utilize max-flow min-cut theorem Ford and Fulkerson (1956) to separate two node sets in a graph by computing a maximum flow and later the minimum cut between the sets. Streaming graph partitioning (SGP) Stanton and Kliot (2012); Tsourakakis *et al.* (2014) is another technique, which targets more recent computing workloads, mainly big-data. These algorithms are faster than the older counterparts but with much less partitioning resolution. Iterative local search Fiduccia and Mattheyses (1982); Kernighan and Lin (1970); Simon and Teng (1997); Karypis and Kumar (1998b) includes a vast volume of techniques by starting from a random solution and refining it through an iterative exchange of nodes between different groups. The exchange policy affects the quality of the final partitioning and the overall convergence time. Since RWNNs resemble a DAG graph, the partitioning of the RWNN graph onto a multi-FPGA pipeline boils down into a directed graph partitioning problem. We can leverage heursitics from previous techniques to address the pipeline throughput and memory efficiency optimizations for mapping RWNNs onto multi-FPGA platforms.

Chapter 3

HETEROGENEOUS PROCESSORS ON THE EDGE

As discussed in Chapter 1.1, OpenCL-based field-programmable gate array (FPGA) computing is a promising technology for addressing the edge-computing challenges. To demonstrate the feasibility of the FPGAs, we study the suitability of deploying FPGAs for edge computing through experiments focusing on the following three perspectives: (1) sensitivity of processing throughput to the workload size of applications, (2) energy-efficiency, and (3) adaptiveness to algorithm concurrency and dependency degrees, which are important to edge workloads as discussed above.

The experiments are conducted on a server node equipped with an Nvidia Tesla K40m GPU and an Intel Fog Reference Design Unit Intel (2017) equipped with two Intel Arria 10 GX1150 FPGAs. Experiment results show that (1) FPGAs can deliver a predictable performance invariant to the application workload size, whereas GPUs are sensitive to workload size; (2) FPGAs can provide 2.5–30 times better energy efficiency compared to GPUs; and (3) FPGAs can adapt their hardware architecture to provide consistent throughput across a wide range of conditional or inter/intra-loop dependencies, while the GPU performance can drop by up to 14 times from the low- to high-dependency scenarios.

Figure 3.1: An Intel Fog Reference Design Unit Hosting Two Nallatech 385A FPGA Acceleration Cards.

3.1 Methodology

To confirm and quantify the aforementioned benefits of FPGA-based edge computing in chapter 1.1, we designed and conducted three sets of experiments to evaluate FPGAs vs. GPUs from the perspectives of (1) *performance sensitivity to workload size*, (2) *adaptiveness to algorithm concurrency and dependency degrees*, and (3) *energy efficiency*.

All the GPU-related experiments were conducted on a server node equipped with an Nvidia Tesla K40m GPU, dual Intel Xeon E5-2637 v4 CPUs, and 64GB of main memory. All the FPGA-related experiments were conducted on an Intel Fog Reference Design unit Intel (2017) (see Figure 3.1) equipped with two Nallatech 385A FPGA Acceleration Cards (Intel Arria 10 GX1150 FPGA), an Intel Xeon E5-1275 v5 CPU, and 32GB of main memory. The OpenCL kernels for FPGAs were compiled using

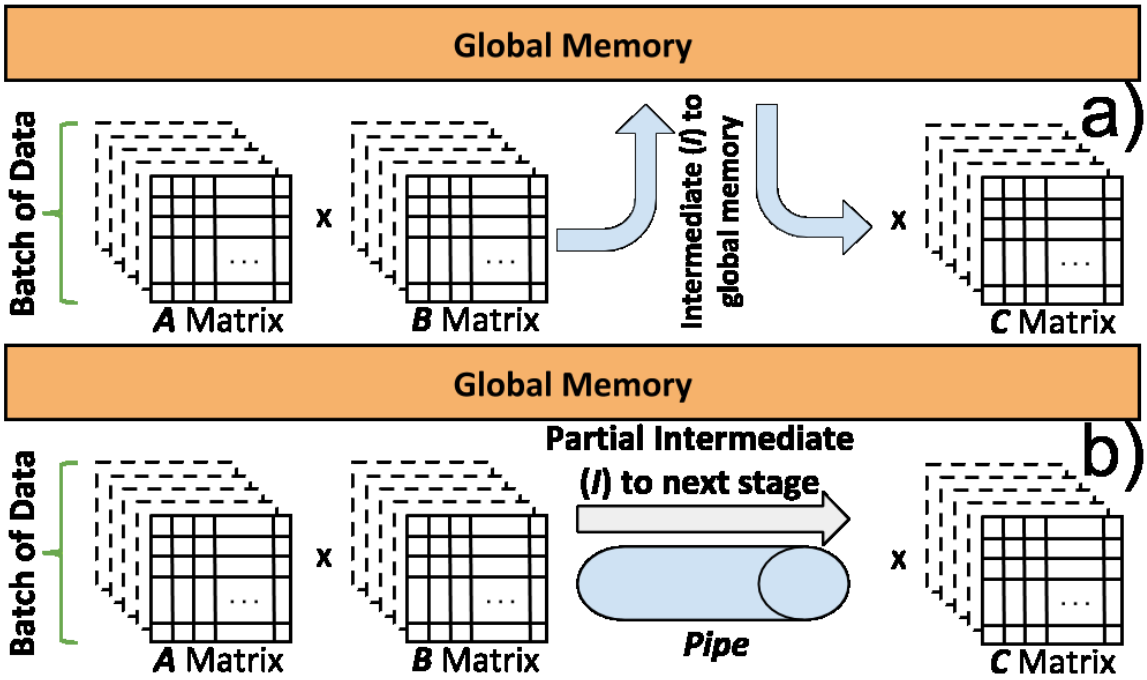


Figure 3.2: Multi-stage Matrix Multiplication on (a) a GPU and (b) an FPGA.

Intel FPGA SDK for OpenCL (version 16.0) with Nallatech *p385a_sch_ax115* board support packages (BSP). The GPU OpenCL kernels were compiled at runtime using available OpenCL library in CUDA Toolkit 8.0. Results discussed in the next chapter will show that the FPGA substantially outperforms the GPU in several important aspects, despite that the GPU has a much higher theoretical throughput (4.29TFlops) than the FPGA (1.5TFlops).

3.2 Experiment Results

3.2.1 Sensitivity to Workload Size

The purpose of this experiment is to demonstrate the sensitivity of FPGA and GPU to workload size. IoT devices are usually latency sensitive and expect predictable latency and throughput from edge servers. We used a two-stage matrix multiplication ($A \times B \times C$) as the benchmark, to model edge workloads. This opera-

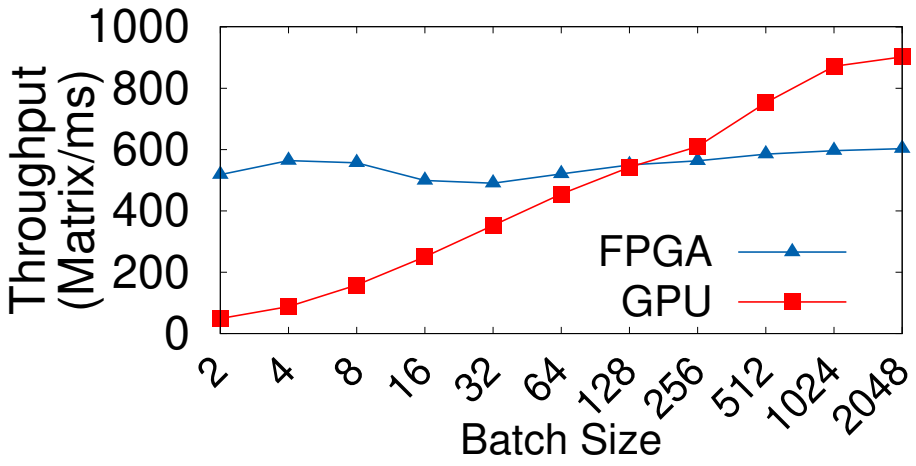


Figure 3.3: Sensitivity of Matrix Multiplication Throughput (Number of Computed Matrices Per Millisecond) Sensitivity to Batch Size (Number of Matrices Received per Batch)

tion is widely used in linear algebraic algorithms and is generic enough for the purpose of this experiment. We believe most IoT workloads, such as voice and image recognition, are heavily dependent on the linear algebraic operations. All three matrices are of dimension 32x32 and contain single-precision floating-point random numbers. Input matrices are provided as a batch, and the batch size represents the workload size. We varied the batch size between 2 to 2048 in the experiment. The processing throughput (number of matrices/ms) is defined as the ratio of the workload size over the total runtime.

Figures 3.2a and 3.2b illustrate the difference of execution flow between the GPU and the FPGA. To exploit spatial parallelism, the GPU must first read the data from DRAM, perform $A \times B$ for the entire batch, and stores the intermediate results (I) in the GPU global memory. Once the writing of I is done, the subsequent $I \times C$ can be performed by reading I back from the global memory. Differently, the FPGA can also exploit temporal parallelism and utilize dedicated *pipes* (*channels*) to transfer the intermediate results from one stage to another without blocking the execution.

Unlike the GPU, the FPGA reads the input from the Ethernet I/O channel. The execution of $A \times B \times C$ is fully pipelined by the streaming architecture implemented in the FPGA, such that the matrix samples can flow in and out of the FPGA through I/O channels one after another without waiting regardless of the batch size.

Figure 3.3 shows the throughput comparison between the GPU and the FPGA across different batch sizes. It is shown that the FPGA can deliver a consistently high throughput by jointly exploiting spatial and temporal parallelism. Specifically, the FPGA outperforms the GPU for small batch sizes (up to 128) in spite of its much lower operating frequency. In contrast, the GPU performance varies largely according to the batch size. GPUs rely on interleaving a large batch of input data to hide the device initialization and data communication overheads. When dealing with small batch size, such overheads will dominate total execution time and degrade the throughput especially when the operations involved have some levels of dependency. Overall, the experiment results imply that FPGAs not only are efficient in handling aggregated service requests coming from individual devices in small batch sizes but also can guarantee a consistently high throughput with a well-bounded latency. Therefore, FPGAs are highly suitable for edge computing given the considerable variance in workload size of various IoT applications.

3.2.2 Adaptiveness

To evaluate how well FPGAs and GPUs adapt to algorithm characteristics, we designed benchmarks to capture two types of dependencies: *data dependency*, which represents the dependency across different iterations of a loop, and *conditional dependency*, which represents the dependency on conditional statements with each iteration of the loop.

Our benchmark resembles an algorithm made of a simple iterative block (*for-loop*) where each iteration performs a certain number of operations. The *loop_length* and *ops* variables define the total number of iterations and the total number of operations per iteration (set to 262144 and 512 in the experiment), respectively. All variables are single-precision in the experiments. Note that the objective of our experiments is to reveal the impact of architecture adaptiveness to algorithm characteristics rather than evaluating the performance for a specific algorithm. In addition, our synthetic algorithm with a single for loop is generic enough to model a large set of computationally intensive applications.

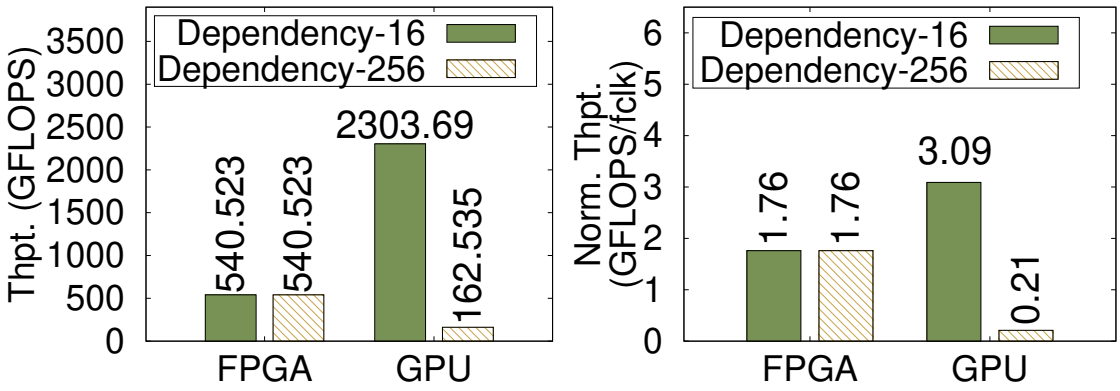
The benchmark captures data dependency by introducing dependency among different iterations of the loop. When there is no data dependency, every single iteration is considered as independent, and all the iterations can execute in parallel. With data dependency, the iterations that are dependent on one another need to be executed sequentially as a group. Therefore, by varying the data dependency degree, i.e., the average size of the groups, we can control the data parallelism available in the algorithm using this benchmark. GPU's performance is closely tied to the available data parallelism. In comparison, FPGA can exploit PEs in series and receive iterations regardless of the dependency. Different iterations can co-exist and be executed in the pipeline while traversing down the connected PEs concurrently.

To introduce conditional dependency, we add *if-else* statements into the iterations of the loop in the benchmark. Half of the iterations are in the *if* block and the other half are in the *else* block. Only the iterations that follow the same branch path can be executed in a data parallel fashion. To reveal the performance impact of conditional dependency, we vary the number of operations in each *if* and *else* block, which affects the initialization overhead and consequently the overall performance. GPU is highly sensitive to conditional dependency because it can parallelize only the iterations that

take the same path at one time. In comparison, FPGA can configure the hardware to include all different execution paths, and use a simple lookup table to direct every thread into the right pipeline and execute all threads at the same time.

In order to get the best performance out of the FPGA and the GPU, the above algorithms were deployed using two different methods. For the GPU, we designed an equivalent OpenCL kernel and deployed it in the NDRange mode to accelerate concurrent operations by exploiting spatial parallelism. For the FPGA, we compiled the FPGA kernel in the *single-threaded* mode to accelerate dependent operations by exploiting temporal parallelism, in which case loop execution is initiated sequentially in a pipelined fashion.

Data Depndency. Figures 3.4a and 3.4b show the raw and the normalized throughput (to system frequency f_{clk}) for both a low (16) and a high (256) data dependency, respectively. In general, computation throughput is linearly proportional to both f_{clk} and architectural parallelism. The normalized throughput decouples f_{clk} from the evaluation and measures the pure impact of architecture parallelism on throughput. For the GPU, the *base* frequency of the board is used as f_{clk}. For the FPGA, f_{clk} is extracted from the full compilation report. It is shown that the GPU performance drops by 14 times from the low to the high data concurrency. As data concurrency increases from 16 and 256, the available data parallelism (the number of loop iterations that can be executed in parallel) for the GPU drops from 16384 to 1024. It is the lack of temporal parallelism that makes GPUs hardly adaptive to such changes in concurrency and dependency degrees. On the contrary, the FPGA delivers a stable throughput regardless of such changes. This is because the hardware resources on an FPGA can be reconfigured dynamically to compose either spatial or temporal parallelism (interchangeable) at a fine granularity. As a result, FPGA outperforms GPU by 3.32 folds with the high data concurrency, and this gap is expected to grow



(a) Raw Throughput

(b) Normalized Throughput

Figure 3.4: Comparison of (a) Raw and (b) Normalized Throughput at Low and High Data Dependency Degrees.

as the dependency degree further increases.

Conditional Dependency Figure 3.5 shows the performance drop with respect to the conditional dependency introduced by *if-else* statements, as the number of operations *if* and *else* block from 8 to 1024. It shows that the FPGA performance is relatively stable as the conditional dependency increases. For some specific cases, the performance is even increased due to a higher clock frequency compared to the baseline kernel. In contrast, the GPU experiences up to 37.12 times performance drop, compared to the baseline kernel with no conditional statements. Branches from the conditional statements cause different threads in a warp to follow different paths, creating instruction replay and resulting in reduced throughput. Figure 3.5 also shows that having fewer operations in the kernel causes more degradation for the GPU since a smaller kernel requires less computation and incurs relatively higher initialization and data transfer overhead.

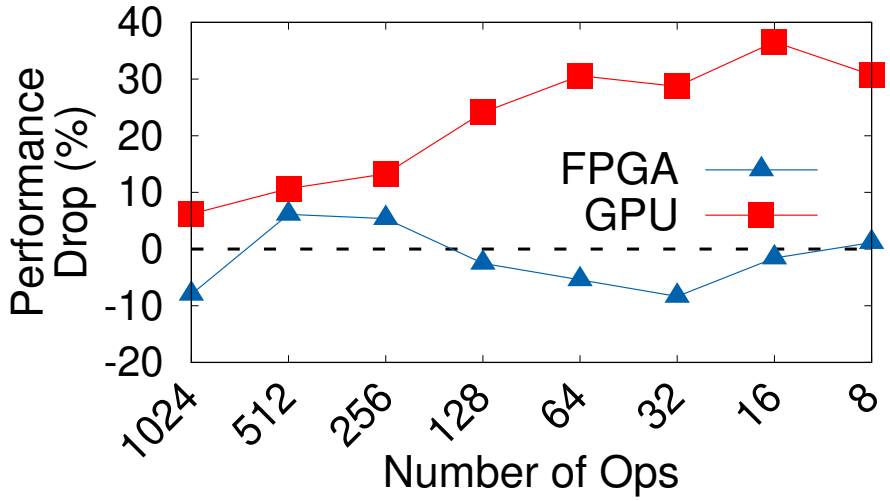


Figure 3.5: Performance Drop Comparison for Kernel with Conditional Statements.

3.2.3 Energy Efficiency

To evaluate energy efficiency, we measured the workload throughput divided by its average power usage. To project energy efficiency, the power consumptions of both devices are recorded for all of the experiments. We used the *nvidia-smi* command-line utility and the Nallatech memory-mapped device layer API to query the instant board-level power consumption every 500 milliseconds for the GPU and FPGA, respectively. We then calculated the average power usage by averaging all the power numbers recorded across five trials of each experiment. Need to mention that our heterogeneous testing platform does not affect the energy calculation since we only measure the board power consumption.

Figure 3.6a and 3.6b show the power consumption and energy efficiency comparison for performing the matrix multiplication tasks mentioned in chapter 3.2.1, for different batch sizes. Running at a much lower frequency, the FPGA consistently consumes 2.79–3.92 times lower power than the GPU. Taking into account the performance, it shows that the FPGA can provide 2.6–30.7 times higher energy efficiency than the GPU for executing matrix multiplication. The improvement is prominent,

especially for small batch sizes. The low power consumption and the high energy efficiency of the FPGA imply that deploying FPGAs for edge computing can potentially gain better thermal stability at lower cooling cost and reduced energy bill.

Figure 3.6c depicts the energy efficiency comparison for running the workloads with different dependency degrees (mentioned in chapter 3.2.2). The results show that the FPGA achieves a similar throughput to the GPU for executing the kernels with a high data concurrency degree (low data dependency degree of 16). For the high-data-dependency (degree of 256) workload, the FPGA achieves up to 11.8 times higher energy efficiency than the GPU. Such energy efficiency improvement is expected to further increase as the dependency degree grows. The experiment results indicate that the FPGA is almost on par with the GPU regarding energy efficiency for executing high-concurrency algorithms, while it can significantly outperform the GPU for executing high-dependency algorithms.

3.3 Conclusions

In this chapter, we studied three general requirements of IoT workloads on edge computing architectures and demonstrated the suitability of FPGA accelerators for edge servers. Our results confirm the superiority of FPGAs over GPUs with respect to (1) providing workload-insensitive throughput; (2) adaptiveness to both spatial and temporal parallelism at fine granularity; and (3) better energy efficiency and thermal stability. Based on our observations, we argue that FPGAs should be considered a replacement or complementary solution for current processors on edge servers.

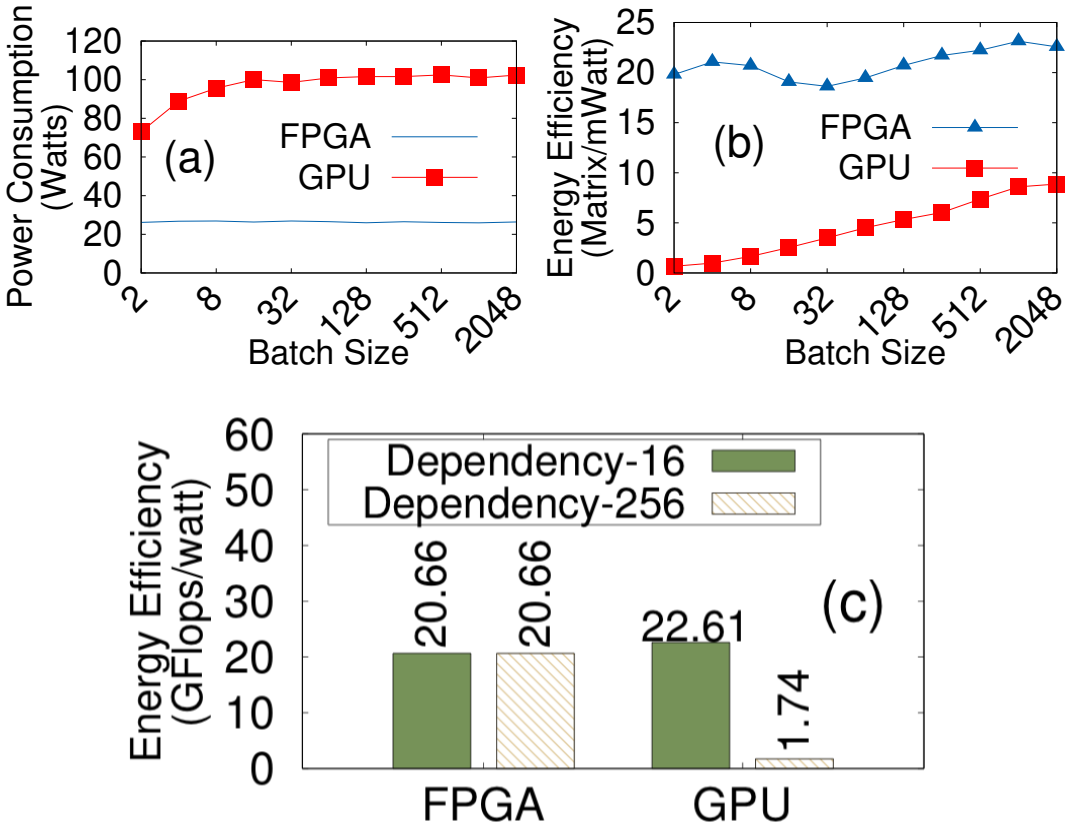


Figure 3.6: The Caparisons of (a) Power Consumption and (b) Energy-efficiency for the Matrix Multiplication Tasks and (c) the Data Dependency Benchmark.

Chapter 4

LOOP ACCELERATION IN HETEROGENEOUS SYSTEMS

As discussed in Chapter 1.2, although a general understanding of different accelerators is available, choosing the right accelerators for applications in a heterogeneous computing system is still a difficult problem. To address this challenge, we study how the accelerators with different hardware architectures can accelerate different types of loops by developing *Loopy*, a collection of five fine-grained loop patterns that commonly exist in real-world applications such as linear algebra, optimization, and data analytics algorithms.

In this chapter we evaluate the performance of important loop patterns on several typical accelerators, through parameterizing the key aspects of the loop patterns, including the type and degree of dependencies, data bit-precision, operational intensity, and size of the iteration spaces.

Loop Pattern	Sample Algorithm/Application
Intra-Dimension Dependency	Linear Algebraic Routines
Diagonal Dependency	Needleman-Wunsch
Conditional Dependency	Kmeans, Single-Source Shortest Path
Anti-Dependency	Floyd-Warshall Algorithm
Half-Parallelism Half-Dependency	K-Nearest Neighbor

Table 4.1: List of Loop Blocks

4.1 Methodology

Our approach to understanding how to choose the optimal accelerator for a given algorithm is by studying the performance characteristics of common loop patterns on GPUs and FPGAs. Following this approach, we designed *Loopy*, a set of abstract and configurable loop blocks, which captures the key loop patterns extracted from real-world algorithms (Table 4.1), and allows flexible testing of each type of loops by varying the following key parameters:

1. *Computational intensity*, which is the total number of computational operations that each iteration of the algorithm performs. In our study, it is defined as the number of multiply-accumulation operations. The computational intensity can affect the size of the pipeline and the number of instructions on both FPGA and GPU. Changing this parameter can show how both platforms performances are susceptible to the amount of computation;

2. *Dependency and concurrency degrees*, which defines how many iterations depends on each other and how many other iterations can be executed separately.

3. *Input data size*, which specifies the total number of floating-point variables that the algorithm processes. The size of the input data can affect the load of computation on a target platform, which can decide the suitability of one device over another.

4. *Variable precision*, which is the bit-width size of the variables in the algorithm. FPGAs and most-recent GPUs have the capability to deliver higher performance for lower bit-precision operations.

Loopy includes optimized implementations of each loop type for GPU and FPGA. The rest of this chapter details each loop type and its GPU and FPGA implementa-

tions, and presents experiments from running them on real devices. While optimizing GPU programming has been well studied, *OpenCL-based FPGA optimization is not well explored and not trivial.* In our discussions, we will also detail how we performed the optimizations for each key loop type.

All GPU-related experiments were conducted on two server nodes with two type of GPUs. One server is equipped with an Nvidia Geforce RTX2080 GPU, dual Intel Xeon E5-2637 v4 CPU, and 64GB of DDR4 main memory (2133MHz). The RTX2080 is a large form-factor GPU, suitable for heavy AI and deep learning workloads. Another server is equipped with an Nvidia Tesla T4, Intel Xeon E5-2650 v3 CPU, and 198GB of main memory. The T4 is a small GPU, suitable for edge servers. All the FPGA-related experiments were conducted on an Intel Fog Reference Design unit, equipped with two Nallatech 385A FPGA Acceleration Cards (Intel Arria 10 GX1150 FPGA), and Intel Xeon E5-1275 v5 CPU, and 32GB of DDR4 main memory (2133 MHz).

The OpenCL kernels for FPGAs were compiled using Intel FPGA SDK for OpenCL (version 19.1) with Nallatech p385a_sch_ax115 board support packages (BSP). The GPU OpenCL kernels were compiled just-in-time at runtime using available OpenCL library in CUDA Toolkit 11.0. For the FPGA, we implemented all the kernels in the single-thread mode and NDRange (multi-threaded) mode. Single-thread kernels on the FPGA typically have much less overhead and can achieve much higher clock frequency rate, compared to multi-threaded kernels. Thus we focus on the results from the single-thread mode execution on the FPGA. For the GPUs, we implemented the kernels in the NDRange mode in OpenCL, which will deploy concurrent threads on the available compute units.

The insights from our study can be generalized, irrespective of our FPGA or GPU choices. Our experiments target the general capability of accelerators in handling common loop patterns, rather than handling specific computational blocks (e.g., ten-

sor processing for deep learning applications). Various generations of FPGAs and GPUs usually differ in their total available resources, such as programmable blocks on FPGAs and processing units in GPUs, which do not affect their general behaviors. When comparing the acceleration achieved by GPU vs. FPGA, we also focus on the general trend, i.e., how the performance changes w.r.t. the key parameters identified above, rather than the absolute performance for a specific configuration. Therefore, our characterization of loop acceleration is generally applicable to GPUs and FPGAs regardless of hardware's specific choices.

4.2 Intra-Dimension Dependency

Definition. This type of loops is usually composed of two or more nested iterative blocks, where each level of iterative blocks is considered a *dimension*. In this pattern there exist a *loop-carried data dependency*, which is a dependency of one iteration on the output of the previous iterations (read-after-write), in one or more dimensions, while at the same time one or more dimensions have no dependency between their iterations. In another word, we can observe both dependency and concurrency in the overall iteration space.

For example, in Algorithm 1, the dependency exists between iterations with the index of i. In this algorithm, updating every element of the array A with the index of i on the first dimension depends on the value of the element with the index of $i-1$. Elements in the second dimension with the index of j do not carry any dependency. In this case, the dependency exists on the dimension with the index of i and the concurrency exists on the dimension with the index of j. Figure 4.1 illustrates the iteration space and the dependency graph of intra-dimension dependent loops. Although in this example, the nested loops have only two dimensions, indexed by i and j.

Simple linear algebraic algorithms Guennebaud *et al.* (2010), such as matrix-matrix (see Listing 4.1) or matrix-vector multiplications are following this type of loop pattern. For example, in matrix-matrix multiplication, each cell of the output matrix can be computed separately (concurrency), while the dot multiplication of one row and one column can only be performed sequentially in a single thread (dependency).

Algorithm 1: Intra-dimension dependency algorithm

$i \leftarrow 1$

$j \leftarrow 1$

for $i \leq n$ **do**

  for $j \leq m$ **do**

    // In our case, func is an FMA operation

    $A[i][j] = func(A[i-1][j], B[i][j], ...)$

  end for

end for

```
type Row = List[Double]
type Matrix = List[Row]

def dotProd(v1:Row,v2:Row) =
v1.zip( v2 ).
    map{ t:(Double,Double) => t._1 * t._2 }.
    // Dependent accumulation (Spatial Parallelism)
    reduceLeft(_ + _)

def mXm( m1:Matrix, m2:Matrix ) =
    // Parallel Row-by-Row multiplication (Temporal Parallelism)
    for( m1row <- m1 ) yield
    for( m2col <- transpose(m2) ) yield
    dotProd( m1row, m2col )
```

Listing 4.1: Matrix-Matrix Multiplication Algorithm

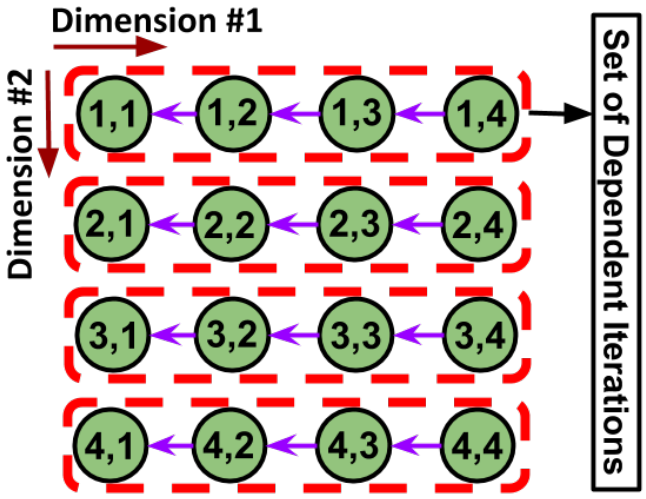


Figure 4.1: Intra-dimension Dependent Loop Pattern.

The degree of spatial and temporal parallelism, combined with the arithmetic intensity, can determine the choice of deployment on either FPGA or GPU. Algorithms with a high degree of dependency can usually finish faster on FPGAs, while algorithms with a high degree of concurrency can utilize the available farm of SIMD compute units on the GPUs and accelerate their execution.

Implementation. Our benchmark contains the GPU and FPGA versions of the intra-dimension dependent loop. For the GPU version, the loop is unrolled spatially over the non-dependent dimension. Each independent iteration is deployed as a work-item (unit of a task in the OpenCL), and the total number of work-items are grouped into several work-groups (unit of execution on a single compute unit). Also, we specifically order the memory access indexes to enable memory access coalescing among work-items in a work-group for better performance. For the FPGA version, we first apply statement re-ordering to place the dependent loop as the inner-most loop, which enables interleaving of the outer-loop iterations (non-dependent) inside the inner-loop cycle. It also helps achieve the initiation interval of one in the inner-most loop. In loop pipelining, the initiation interval is the number of clock cycles

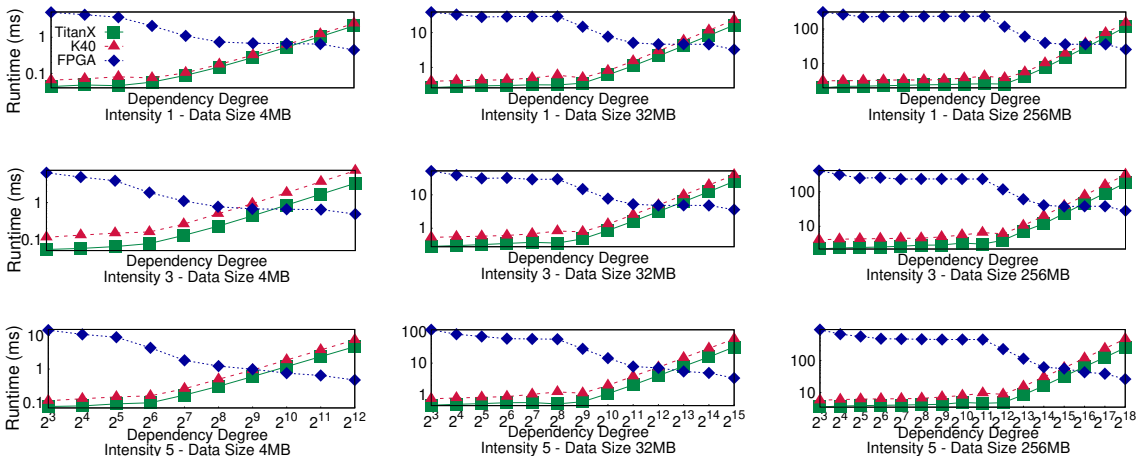


Figure 4.2: Intra-dimension Dependency Performance on the GPU and the FPGA

between the start times of consecutive loop iterations. Having an initiation interval of one enables the FPGA to push one iteration into the pipeline at every clock cycle and achieve the highest performance, which is the ultimate goal for every design. Further, we apply *loop blocking* (also known as loop tiling) on the outer for loop. Doing so enables utilization of the on-chip registers on the FPGA (with the same size of the block), by copying the required data for the execution of the block, as a whole, onto the allocated on-chip registers, thereby reducing the DRAM access overhead.

Experiment. We deployed FPGA and GPU kernels, resembling Algorithm 1. Input data is an array of floating-point variables of a specific size (4, 32, 256 MB). Every single iteration in the algorithm is responsible for a single element in the array. As a result, the total number of iterations is equal to the number of input values. As shown in the algorithm, the dependency and concurrency degrees are configured by changing the number of iterations, n and m, respectively. Figure 4.2 shows the runtime of this intra-dimension dependent loop on both FPGA and GPU.

We can make several key observations from the results. First, GPU does excel at accelerating the loop with a high degree of concurrency. More concurrency can lead to better spatial parallelization, which makes the GPU a great candidate for deployment.

In contrast, with the increase in the dependency degree, the FPGA can take advantage of the configured long pipeline and parallelize the dependent iterations. In this case, with a high degree of dependency, the FPGA can outperform both RTX2080 and T4 up to 21.5x and 13.6x. With a high degree of concurrency, both RTX2080 and T4 perform better than the FPGA, by up to 184x and 93x, respectively.

The second observation is about the effect of computational intensity (the total number of computational operations in each iteration) on the final performance. Higher intensity means more computations, which leads to more pipeline stages. With more pipeline stages, FPGA can handle more dependent iterations and achieve higher performance. Need to mention, the available hardware resources on the FPGA are limited and may block developers from configuring a large number of pipeline stages. As a result, developers may need to adopt a smaller loop block size, which leads to the reduction of the performance. Compared to FPGA, the GPU has to spend more time executing each loop iteration, with no opportunity for pipelining the iteration. For example, Figure 4.2 shows that going from the intensity of 1 to 5, the performance drops by up to 2.1x and 3x, on RTX2080 and T4, respectively.

The third observation is the performance reduction of FPGA for kernels with low dependency, because there are not enough dependent iterations to fully saturate the configured pipeline. In this situation, developers may want to switch into the NDRange mode kernels, which can interleave the parallel iterations into the pipeline and keep it saturated. In comparison, GPU can utilize the massive farm of cores to exploit a high degree of parallelism when the dependency is low. Therefore, as shown in Figure 4.2, FPGA's performance is worse with lower dependency degree whereas GPU's performance is not affected.

4.3 Diagonal Dependency

Definition. Diagonal dependent loops are following almost the same pattern as intra-dimensions dependent loops, except that the dependency is diagonal instead of horizontal or vertical in the iteration space. As illustrated in Figure 4.3, horizontal (vertical) dependency refers to the dependency of an iteration on the left (top) neighbor iterations with the same i (j), respectively. For example, in the aforementioned intra-dimension dependency, there is horizontal dependency among the iterations as shown in Figure 4.1. Diagonal dependency means that an iteration depends on its relative top-left iteration which has both different i and j indexes. For example, in Figure 4.3, iteration $(2, 2)$ depends on its diagonal neighbor iteration $(1, 1)$. Algorithm 2 shows an example of this kind of loops, where the computation requires data from its diagonal neighbor in the iteration space. In specific cases, the dependency can be extended and include either horizontal or vertical, as well.

Parallelization of these types of loops on SIMD architectures, such as GPU, is not straightforward. Depending on the type of diagonal dependency, developers can either parallelize the diagonals or use the wavefront technique Belviranli *et al.* (2015) for parallelization. In the wavefront parallelism mode, kernels are enqueued back to back to the GPU, each computing one set of independent iterations. The number of the kernels is equal to the length of the diagonal.

Dynamic programming algorithms are usually composed of diagonal dependent iterations. A specific example of such algorithm is Needleman-Wunsch Needleman and Wunsch (1970) (see Listing 4.2), which performs matching between two input strings while minimizing the penalty.

```
val alignMat = new Array[Array[ResultEntry]](length)
val contant = ...
val gapPenalty = ...
def getScore(i: Int, j: Int): Score = {
    if (alignMat(i)(j) != null) {
        alignMat(i)(j)
    } else {
        // 3-Way Diagonal Dependency
        val tryMatch = getScore(i - 1, j - 1) +
                       constant
        val horizontalGap = getScore(i, j - 1) +
                       gapPenalty
        val verticalGap = getScore(i - 1, j) +
                       gapPenalty
        if (m == tryMatch) {
            alignMat(i)(j) = (m)
        } else if (m == horizontalGap) {
            alignMat(i)(j) = (m)
        } else {
            alignMat(i)(j) = (m)
        }
        m
    }
}
```

Listing 4.2: Needleman-wunsch Algorithm Score Calculation

Implementation. For the GPU implementation, the parallelization method depends on the existence of vertical or horizontal dependency. In the absence of both of these dependencies, each thread can take care of one diagonal, in parallel. The existence of any of the mentioned dependencies (in addition to diagonal dependency) would

Algorithm 2: Diagonal depedency algorithm

$i \leftarrow 1$

$j \leftarrow 1$

for $i \leq n$ **do**

  for $j \leq m$ **do**

    $A[i][j] = func(A[i-1][j-1], B[i][j], ...)$

  end for

end for

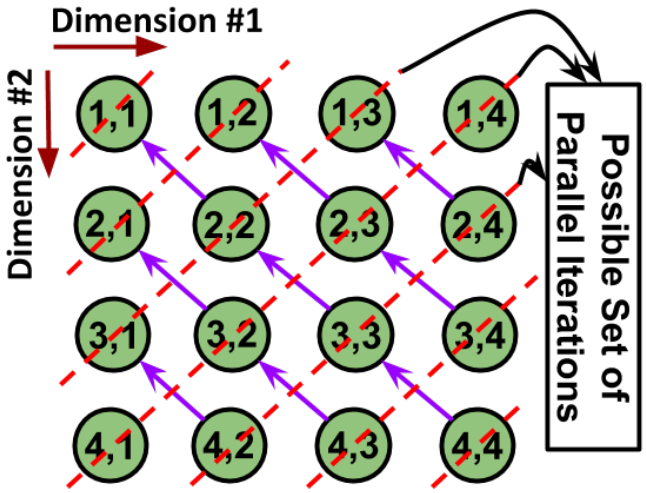


Figure 4.3: Diagonal Dependency Loop Pattern

force the GPU to perform *anti-diagonal parallelization*. As shown in Figure 4.3, the independent iterations that can be parallelized form a line that is perpendicular to the diagonal dependent iterations.

For the FPGA implementation, we first perform loop blocking on the first dimension, which enables caching of the input data for each iteration of the second dimension's iterations. Later, we copy the required data for the second dimension's computation into the allocated on-chip registers of the size block. Every iteration of the second dimension first reads the data from the registers, performs the calculation, and writes back the data to the registers and the DRAM. To handle all elements in the block, each iteration of the second dimension contains a nested loop of size block,

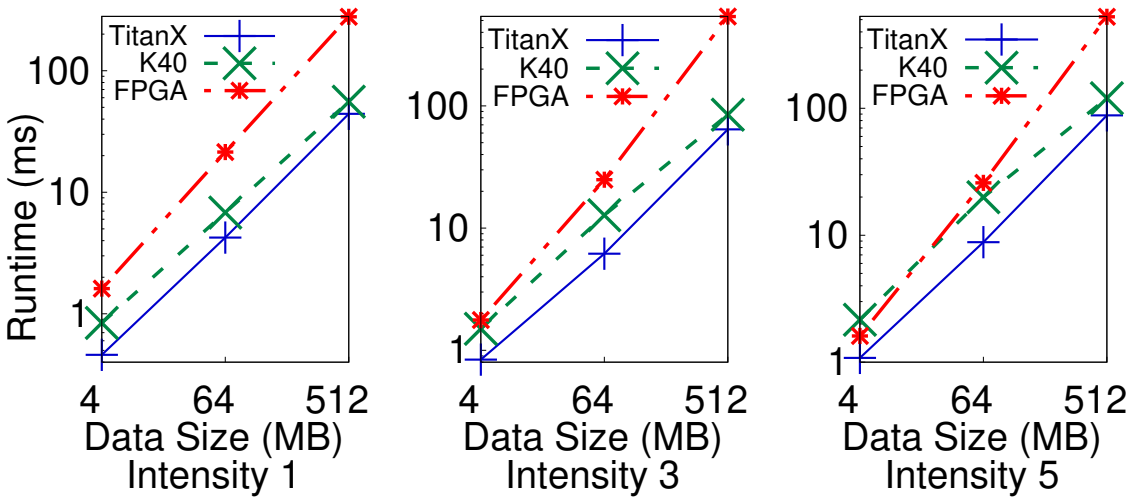


Figure 4.4: Diagonal Dependency Runtime on Both FPGA and GPU. The Dependency is Only Diagonal.

which is fully unrolled. In this implementation, the iterations of the second dimension have a loop-carried data dependency. Unfortunately, the compiler cannot infer an initiation interval of one for this loop body, due to the existence of large latency between consecutive iterations of the loop. To overcome this issue, we interleave the execution of the block iterations inside the second dimension loop, which enables full exploitation of the available pipeline stages. Doing so reduces memory accesses and leads to higher operating frequency and fewer stalls in the pipeline.

Experiment. Figure 4.4 shows the performance of the diagonal dependent loops on the FPGA and the GPUs, where the dependency only exists diagonally. We did measurements for three different computational intensities (1, 3, and 5) and three different input sizes (4, 64, and 512 MBs). The results show that the GPU outperforms the FPGA in almost all cases, except for the experiment with high computational intensity and small data size. In this type of dependency, GPU can assign one diagonal set of iterations to one work-item and exploit high degree of parallelism on all the available cores. In this case, RTX2080 and T4 outperform the FPGA by up to

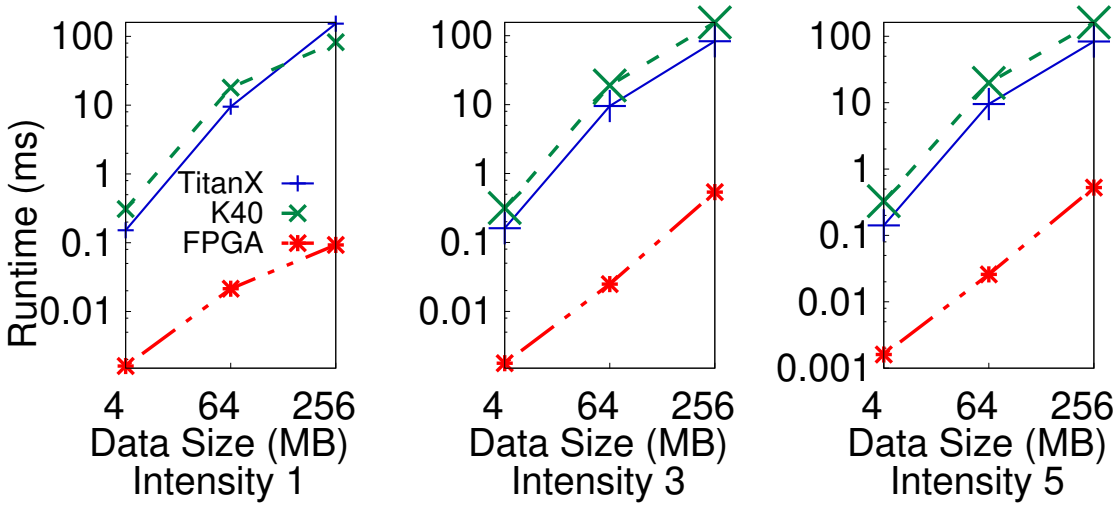


Figure 4.5: Diagonal Dependency Runtime on both FPGA and GPU. The Dependency Also Includes Horizontal and Vertical.

6x and 4.3x, respectively.

Figure 4.5 shows the performance of the same loop pattern but with additional horizontal and vertical dependencies between the iterations. We modified the function f in Algorithm 2 to include both $A[i-1][j]$ and $A[i][j-1]$, in addition to $A[i-1][j-1]$, as its parameters to introduce these dependencies between $A[i][j]$ and its horizontal, vertical, and diagonal neighbor iterations. In this case, the FPGA can utilize the same pipelining method to accelerate the execution, while both GPUs need to use wavefront parallelism and parallelize computation for each anti-diagonal. Unlike the case with diagonal dependency, the wavefront parallelism model cannot exploit a large number of parallel threads. In addition, it needs to repetitively deploy the same kernel to calculate a new set of anti-diagonal iterations. As a result, the FPGA outperforms both RTX2080 and T4 by up to 165x and 322x, respectively.

4.4 Conditional Dependency

Definition. The existence of conditional statements in loop bodies can alter the extent of parallelization on certain accelerators. In loops with a conditional statement, every iteration diverges in the execution path, depending on the specific conditions. Algorithm 3 represents an example, where every iteration performs either the first or the second statement based on the content of an array in that specific iteration index.

Algorithms such as K-means and single-source shortest path (SSSP) consist of many conditional decisions. In the K-means (see Listing 4.3), the clustering of the observations requires many comparisons, based on the distance; SSSP relies on the sparse matrix multiplication, where the number of iterations for each output calculation is non-deterministic.

Algorithm 3: Conditional dependency algorithm

$i \leftarrow 1$
for $i \leq n$ **do**
  if $B[i] > 0.0f$ **then**
    $A[i] = f(B[i], D[i], ...)$
  else
    $A[i] = f(C[i], D[i], ...)$
  end if
end for

```
def run(xs: List[Point]) = {
    var centroids = xs take n

    for (i <- 1 to iters) {
        centroids = clusters(xs, centroids) map average
    }
    clusters(xs, centroids)
}

def clusters(xs: List[Point], centroids: List[Point]) =
    (xs groupBy { x => closest(x, centroids) }).values.toList

def closest(x: Point, choices: List[Point]) =
    // Calculating mininum, which requires several comparisons (if-
    else)
    choices minBy { y => dist(x, y) }

def dist(x: Point, y: Point) = (x - y).modulus
def average(xs: List[Point]) = xs.reduce(_ + _) / xs.size
```

Listing 4.3: Kmeans Algorithm

Implementation. The conditional dependency is introduced by an if-else statement in the kernel. On the GPU, the loop is simply parallelized on different cores, and each thread performs the if-else comparisons. But the SIMD architecture in the GPU cannot efficiently handle the conditional statements in the work-items, due to thread divergence issue. In the FPGA implementation, the kernel is developed in a single-thread mode and the loop is unrolled to the limit of the FPGA area and available DRAM bandwidth. In contrast to the GPU implementation, FPGAs can handle numerical conditional statements, using look-up tables and a simple multiplexer. More

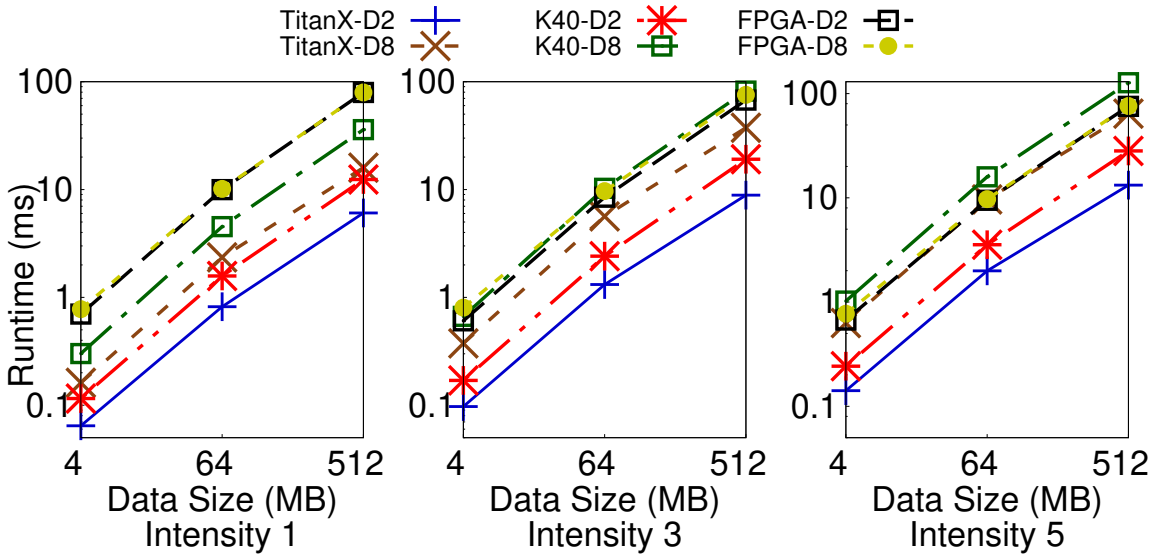


Figure 4.6: Conditional Dependency Runtime on Both FPGA and GPU, for Different Intensities.

specifically, the FPGA can map all different paths of the execution in the design and enable different threads running simultaneously in different conditional blocks.

Experiment. Figure 4.6 shows the runtime of the conditional dependent loop on the GPU and FPGA, with various computational intensities (one, three, and five) number of conditional branches (two and eight) within each iteration as well as various total input data sizes (4, 64, and 512 MB). The number of conditional statements is represented as D2 and D8, for two and eight conditional decisions, respectively. The results show that the FPGA can sustain the same performance among kernels with different conditional branches, whereas the GPU suffers more performance degradation for kernels with more conditional branches (up to 45% slowdown). As a result, the FPGA outperforms the GPU with a higher number of conditional dependencies; e.g., 40% better for a dependency level of eight. This observation suggests the suitability of FPGAs for algorithms with a high degree of decision making during the execution. These types of applications usually cannot exploit the massive parallelism

in SIMD architectures and can be better handled by reconfigurable processors such as FPGAs.

4.5 Anti-dependency

Definition. In this loop pattern, every iteration consists of more than one statement. Unlike the intra-dimension dependent loops, where the dependency is read-after-write, this pattern carries write-after-read dependency. In this pattern, one statement of an iteration reads a data item that is going to be updated by the other statement in the next iteration. It is named anti-dependency because the statements in different iterations are following the write-after-read pattern, as opposed to read-after-write in the typical dependency patterns. Algorithm 4 demonstrates a general example of such loops. The existence of read-after-write dependency creates an anti-dependent loop pattern.

Anti-dependent loops have a unique characteristic. It is possible to face race condition in case of parallelization of all the iterations. More specifically, the first iteration reads the old value of an array element (e.g., $A[i]$ depends on $B[i+1]$ in Algorithm 4), while the second iteration updates the same value, and so on and so forth. When these iterations are executed on different threads to achieve parallelism, the dependent read and write might be executed out of order, which damages the correctness.

Algorithm 4: Anti dependency algorithm

$i \leftarrow 1$

for $i \leq n$ **do**

$A[i] = B[i+1] + C[i] * D[i]$

$B[i] = B[i+1] - E[i] * D[i]$

end for

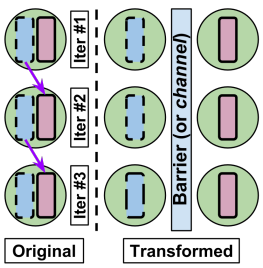


Figure 4.7: Anti Dependency Loop Pattern.

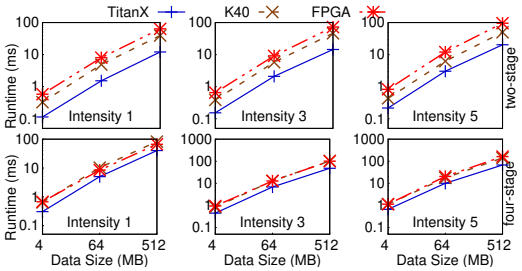


Figure 4.8: Anti Dependency Results for Two and Four Stages.

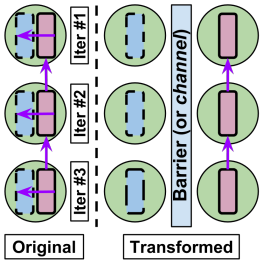


Figure 4.9: Half-parallellism Half-dependency Loop Pattern.

Implementation. These types of loops can be parallelized on vector processors with a global barrier mechanism among all SIMD threads. Unfortunately, both the FPGA and the GPU lack such a global barrier mechanism between all threads. An approach to parallelizing inter-iteration dependent loops is loop-splitting. In this approach, the loop can be divided into multiple separate loops, where none of them carries any dependency. In this situation, loops should run sequentially on the target processor (to guarantee the correctness of the execution), but each loop can fully exploit the available spatial core units. Figure 4.7 represents the execution and the dependency of the original loop, along with the transformed version of it. The dotted blue box and the solid red box represent different statements in the loop body. The arrow shows the anti-dependency between different statements of consecutive iterations.

To accelerate anti-dependency loops on GPU and FPGA, we apply statement re-ordering and loop splitting. The transformation creates multiple flattened loops, where each of them represents a stage of the execution. The lack of global barriers prevents both platforms from co-locating the execution of the generated sub-loops after the main loop distribution, except for using channels in FPGA, which is a mechanism for passing data between kernels and synchronizing kernels with high

efficiency and low latency. Usually, kernels need to communicate through DRAM, which increases the application runtime. By using channels, loops can start pipelining their partial results to the next loop, which enables co-location of the computation and communication and reduces the application runtime. Unlike the FPGA, GPU should execute the flattened loop sequentially, but each stage can be fully parallelized spatially.

Experiment. Figure 4.8 shows the runtime of the FPGA and GPU in accelerating these loops. We varied the degree of anti-dependency which is the number of statements involved in the anti-dependency. For example, in Algorithm 4 the dependency exists between two statements, which yields into the anti-dependency degree of two. As a result, the main loop in the benchmark can be split into several separate and parallelizable loops, depending on the number of anti-dependent statements in the loop body. We also varied the intensity level and input data size.

Comparing the runtimes for the case of four stages of anti-dependencies, the FPGA can outperform the Tesla K40 GPU for kernels with low intensity (up to 20% speedup), whereas it performs close to the GPU for higher intensities (up to 15% speed degradation). Comparing to Titan X, the FPGA performs 1.6x slower. Kernels with higher intensities lead into larger area consumption and limit the parallelism level in each stage, which further results in the reduction of the channels widths. Overall, increasing the number of statements with anti-dependencies will result in more separate loops. As shown in Figure 4.8, increasing the degree of anti-dependency reduces the gap between the FPGA and GPU. We can expect that by following this trend, the FPGA will eventually outperform the GPU.

4.6 Half-Parallelism Half-Dependency

Definition. Half-parallel half-dependent loops usually include the dependent and the parallel statements, simultaneously, and consist of only one loop, with no nested loop. Algorithm 5 lists an example of this type of loops. The existence of loop-carried dependent statements (read-after-write) prevents the spatial parallelization of the algorithm, as a whole. Transforming the loop into multiple flattened loops enables the execution of the loop in two different stages. Unlike the anti-dependent loops, the loop-splitting process does not enable spatial parallelization opportunity for all the loops, since part of the algorithm carries read-after-write dependency. After the splitting, the parallel portion of the loop can be deployed on processors with a high number of parallel compute units, e.g., GPUs, while the dependent portion can be handled by processors that are suitable for sequential execution, e.g., on CPUs and FPGAs.

Figure 4.9 represents the half-parallelism half-dependent loop pattern. For this pattern, each red box in an iteration depends on another red box from the previous iteration. Furthermore, each red box depends on the value of the blue box in the same iteration.

Half-parallel half-dependent applications such as K-nearest neighbor (KNN) include of both parallel parts (distance computation) and dependent parts (sorting). These applications can utilize one or more hardware accelerators for an efficient acceleration.

```
val sortedDistances = data.map{case (a, b)
        => (b, Util.euclideanDistance(p, a))}
        .sortBy(_._2, ascending = true)
// take the top k results
val topk = sortedDistances.zipWithIndex()
        .filter(_._2 < k)
// take the most predominant class within the top k
val result = topk.map(_._1)
        // Parallel section of the KNN
        .map(entry => (entry._1, 1))
        .reduceByKey(_+_)
        // Semi-Dependent section of the KNN
        .sortBy(_._2, ascending = false).first()
```

Listing 4.4: KNN Algorithm

Implementation. We apply loop splitting to separate the parallel section from the dependent section. For the GPU, we first compute the parallel part on the GPU and then transfer the data back to the main memory of the host and execute the dependent part on the CPU. Running the dependent block of code on the GPU is not efficient and will lead to poor performance. For the FPGA we have multiple options, (1) running the parallel and dependent blocks of the loop serially on the FPGA, (2) running the parallel block on the FPGA and the dependent block on the CPU, and (3) using channel to pipeline the intermediate result from the parallel part to the dependent part and decrease the running time overhead. Using the channels is the

best available option to co-locate computation and communication and achieve the highest possible performance.

Algorithm 5: Half-parallelim Half-dependency Algorithm

$i \leftarrow 1$

for $i \leq n$ **do**

$A[i] += C[i] * D[i]$

$sum += B[i] + A[i] + D[i]$

end for

Experiment. Figure 4.10 shows the runtime of the FPGA and the GPUs in acceleration these loops. For this experiment, we provided input data with a size of 1 to 1024 MB. The FPGA can outperform both Titan X and Tesla K40 GPUs, by up to 118x and 110x, respectively. The overhead of the data transfer from the GPU to CPU reduces both GPUs' performance significantly. As a conclusion, co-locating the parallel and the dependent sections of the code on the FPGA can yield much higher performance, compared to utilizing GPU+CPU combination with a much slower communication channel.

4.7 Conclusions

In this chapter, we designed Loopy for studying common loop patterns on important GPU and FPGA accelerators. We identified and analyzed five common loop patterns, along with the key configuration parameters in these patterns. We then studied the acceleration opportunities for these loop patterns and how the loop configurations and accelerator platforms affect the effectiveness of acceleration. Using Loopy, developers can gain a good understanding of the acceleration potential of their algorithms on different platforms, without having to implement them for any specific platform, based on the loop patterns that these algorithms embody. LoopBench is

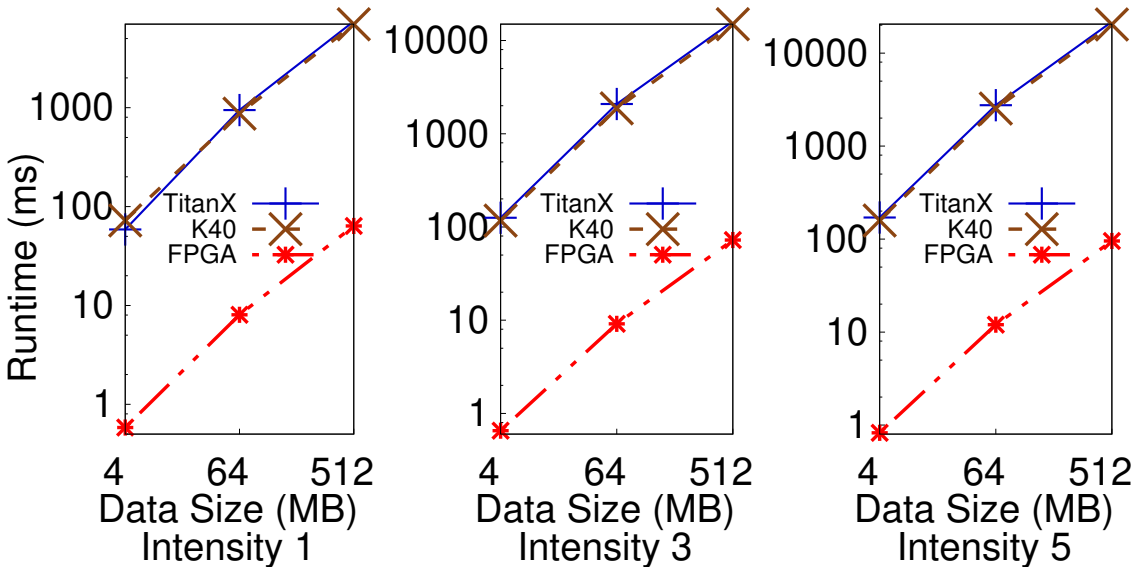


Figure 4.10: Half-parallelism Half-dependency Runtime on Both FPGA and GPU, for Different Intensities.

open source and publicly available.

Loopy provides an important first step towards the optimized use of accelerators for diverse applications in heterogeneous computing systems. Based on Loopy, we will be able to study the combination of the loop patterns and their accelerations ability in heterogeneous computing systems in our future works.

Chapter 5

MULTI-FPGA ACCELERATION FRAMEWORK

As discussed in Chapter 1.3, commodity hardware accelerators lack the potential to guarantee low-latency services for individual requests. In contrast to widely-used accelerators, FPGAs can leverage their reconfigurable deep pipeline to service the requests in a streaming fashion and provide a predictable low latency. But even with the power of the FPGAs, the ever-increasing complexity of emerging CNNs requires FPGAs with a higher amount of resources, such as memory bandwidth and logical units, to achieve low-latency and high-throughput inferences. This challenge can be potentially addressed by utilizing a cluster of FPGAs, connected through a high-bandwidth communication infrastructure.

In this chapter, we present a novel multi-FPGA CNN accelerator that can leverage a deep pipeline of FPGAs, connected through a high-performance I/O channel. To demonstrate the feasibility of our accelerator, we performed multiple experiments using different widely-used CNN models.

Through this chapter, we discuss our design and we evaluate its effectiveness, compared to other available solutions.

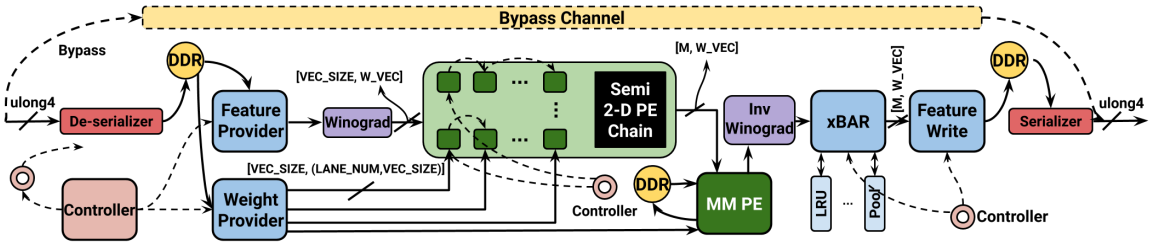


Figure 5.1: CNN accelerator architecture.

5.1 Methodology

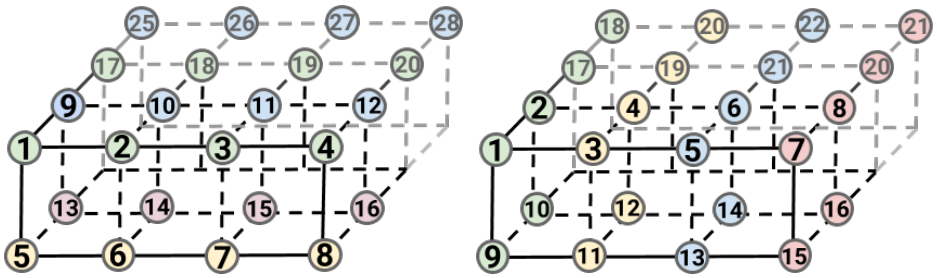
In this chapter, we discuss the overall architecture of our CNN accelerator and the host side manager for distributing and accelerating of a target neural network model. More specifically, first, we discuss the anatomy of the basic building blocks of our CNN architecture. Second, we describe how this CNN architecture is further extended to support 3D convolutions for use cases such as video processing. We also explain our method for choosing the best strategy for mapping 3D convolution onto our native 2D convolution accelerator. Third, we describe the multi-FPGA support for our design architecture. Lastly, we discuss our algorithm for the efficient mapping of various layers of a CNN onto the available chain of the FPGAs.

5.1.1 CNN Accelerator Architecture

Our CNN accelerator skeleton has adopted the 1D systolic array architecture from DLA Aydonat *et al.* (2017). We applied various optimizations on the DLA to achieve higher performance and lower resource utilization. We also extended it to support 3D convolutions and data transmission with other FPGAs. Figure 6.5 depicts the overall architecture of our CNN accelerator design. Our design consists of several key components. First, the *Controller* acts as the coordinator between all the other components. The *Controller* sends the respective configuration parameters to the *Feature Provider*, *Processing Elements (PEs)*, etc. These parameters are usually the type of the layer, size of the input data for processing, and all other related necessary parameters for executing a layer. Second, the *Feature Provider* is responsible for reading the data in a fixed size and feeding it into the first *PE*. This component reads the data from the global memory, caches it in the respective local memory, and sends it over a channel to the first PE in the systolic array, which is a grid of connected

PEs. The channel is a communication medium between two components in the same kernel or different kernels on separate FPGAs. Third, the *Weight Provider* takes care of updating the weight buffers in all the PEs. Since the number of the PEs is usually less than the number of output channels, the PE needs to update the weight buffers multiple times, while handling a single layer. Fourth, the *Cross Bar (xBar)* and all the attached *activation layers* apply the required transformation (pooling, ReLU, etc.), after each convolution or matrix multiplication for each layer. Fifth, the *Feature Writer* receives the output from the *Cross Bar* and writes it back to the global memory. Latter layers further use this data. Sixth, the *Serializer* and the *Deserializer* are responsible for receiving/sending the intermediate results from/to the previous/next FPGA in the chain of the FPGA cluster, respectively. Seventh, the Winograd and inverse (inv) Winograd convert the data into Winograd format and revert the data into the normal representation, respectively. Finally, the controller activates the *Bypass Channel* if the FPGA handles a sub-section of a layer (sub-layer), instead of a whole layer or a group of layers. This channel is responsible for bypassing the partial results from processing a sub-layer to the next FPGA and potentially to the FPGA that handles the last sub-layer of a specific layer. The FPGA that handles the final sub-layer concatenates all the partial results and generates the complete output for that layer. Need to mention that the *Bypass Channel* is de-activated if the FPGA is handling the first of the last sub-section of a layer.

Feature Provider The feature provider is responsible for reading the input data from the global memory and feeding it into the first PE. Further, each PE forwards the received data to the next available PE in the chain. Reading data from the global memory is critical in the accelerator design. Non-optimized data read from the global memory can lead to major stalls (low throughput) in the pipeline. To maximize the throughput, our design leverages two main optimizations: (1) memory



(a) Logical View of the Traditional Data Layout.

(b) Logical View of the New Data Layout.

Feature Data Address	1	2	3	4	5	6	7	8	9	...	32
DRAM Data Address	0x00	0x08	0x01	0x09	0x02	0x10	0x03	0x11	0x04	...	0x31

(c) Traditional Data Layout Arrangement on the DRAM.

Feature Data Address	1	2	3	4	5	6	7	8	9	...	32
DRAM Data Address	0x00	0x01	0x02	0x03	0x04	0x05	0x07	0x08	0x09	...	0x31

(d) New Data Layout Arrangement on the DRAM.

Figure 5.2: Traditional Feature Data Arrangement vs. New Data Arrangement. In Traditional Arrangement, the Data is First Stored by Rows and Then the Input Channels. In the New Arrangement, for Each Row We Store a Aet of Input Channels, Sequentially.

data access coalescing, and (2) caching data in the local memory. First, every access to the global memory requires hundreds or thousands of clock cycles. At the same time, the global memory can provide only a chunk of data in every memory access. As a result, to reduce the overall memory waiting time, it is critical to reducing the memory access frequency by coalescing multiple load/store requests. By doing so, the *Feature Provider* can almost saturate the external memory bandwidth, and consequently, minimize the total number of memory transactions. Second, the local memory can be up to a hundred times faster than global memory. Hence, caching

and re-using data on the local memory can provide significant performance benefits.

For efficient global memory data access, we applied data rearrangement on the input data. Previous works Aydonat *et al.* (2017); Zhang *et al.* (2018) demonstrated the importance of memory interface bit-width, which is the total number of bits that can be accessed in one memory transaction, and burst length, which is the total amount of data that is going to be fetched from memory in multiple sequential memory accesses, to performance. For example, for the Intel Arria 10 FPGAs, a minimum bit-width of 512 and a burst length of above 128KB are required to saturate the 16GB/s per bank memory bandwidth. The traditional row-major data representation (used in DLA Aydonat *et al.* (2017)) has limitations to achieve efficient burst-length due to discontinuous DRAM access. To alleviate this problem, we propose a partial input-channel-major data arrangement. Figure 5.2 represents both the traditional and the new data arrangement. In this new method, the host divides the input channel section into multiple chunks of size *VEC_SIZE*. Further, it iterates over the data in a row-major manner, but instead of storing a single data item, it stores the whole *VEC_SIZE*. This data rearrangement is applied to the input data before it is streamed into the FPGA.

The *Feature Provider* needs to send the data in the right format and size to the PEs, so that they can perform the convolution correctly. Starting from a convolution, each PE has to convolve a collection of weight parameters for a single output channel, with a section of the input data, which has the same width, height, and input channel size. We call this piece of data a *brick*, with a size of $Width \times Height \times IN\_CH\_SIZE$. Due to the utilization of Winograd, the $Width$ is always equal to W_VEC (in our case, it equals eight). The *Feature Provider* does not send the whole brick at once, but instead sends it in the granularity of $Width \times 1 \times VEC\_SIZE$, which we call a *plate*. Each plate has a width of eight, a height of one, and a depth of VEC_SIZE. The total

number of plates in each brick is equal to $Height \times (IN\_CH\_SIZE VEC\_SIZE)$. Due to the arrangement of the data in the global memory, reading each plate or multiple sequential plates leads to fully sequential data access in the memory. Sequential data access helps reading more data in fewer transactions and leads to better utilization of the memory bandwidth. The new data arrangement places the required data sequentially in the memory and enables the *Feature Provider* to load the required data with the minimum number of memory accesses.

Weight Provider The *Weight Provider* updates the weight buffers on all PEs while servicing a layer. In our design, each PE generates all the particular features for a single output channel. Since the total number of PEs (it is 32 in our design, based on the available DSPs) is usually less than the actual number of output channels (up to 8096 in different models), they can only generate the features for a certain number of output channels (32). For the rest of the outputs, the controller needs to refresh the PEs with the new set of weight parameters and initiate the same process as the previous round. Finally, similar to the *Feature Provider*, the Weight Provider should maximize the DDR bandwidth (the available data transfer rate between the global memory and the weight and input providers) to minimize the overall stall, while reloading the weight buffers on the PEs. As a result, the weights are reordered, similar to the input data (Figure 5.2), to enable efficient burst-length and bit-width.

Processing Element (PE) PEs are the main building blocks of the design, for the computation of the convolutions and the matrix multiplications. In our design, PEs are all arranged in a semi-1D systolic array fashion. We call it semi-1D, since it adopts the original 1D systolic array, while the outputs from the PEs are forwarded in a 2D fashion for area optimization purposes. Figure 5.3.a represents the general architecture of the array of PEs. Each PE has a dedicated channel to the next PE in the same column, which forwards the arrived input data (a plate) to the next PE. In

contrast, the first PE of each column also forwards the data to the next first PE of their neighbor column, so that each column can have access to the input data. Doing so helps PEs to avoid reading the same piece of data directly from memory, which leads to significant performance and area overhead. The *Feature Provider* streams the data only to the first PE of the first column, and the PEs transmit the same piece of data to the other PEs. Doing so reduces the effort for the wiring between the *Feature Provider* and the PEs, and makes the process of fitting the model on the FPGA more manageable during the compilation.

Our novel semi-1D systolic array architecture is mainly designed to reduce the resource consumption of the connections between every two consecutive PEs. In a traditional systolic array architecture, the total number of wires between PEs i and $i+1$ is equal to $(i+1) \times W\_VEC$. As we go through the PEs in the systolic arrays (increasing the i) we observe a higher number of wires consumption. The total number of wires for an architecture with P PEs will be equal to $P * (P+1))/2 * W\_VEC$. In the semi 1D architecture, we arrange the PEs in a grid fashion, with n rows and m columns ($n \times m$ = P), as shows in Figure 5.3. In this architecture, the number of wires between each two PEs in a row is following the same pattern as the traditional design, while going from one row to another resets the value of i to 1, which significantly reduces the number of wires. We have some extra wiring between PEs in a column for bypassing the computed data. In each column, the total number of wires between the PEs with indexes (i, j) and $(i, j+1)$ ((i, j) is the index of the PE in the grid) is equal to $(j+1) \times W\_VEC$. As a result, the total number of wires in the design is $C \times m + D$, where C is total number of wires in a column and is given by $(n \times (n+1))/2 \times W\_VEC$, and D is the total number of output wires in the last row and is given by $(m \times (m+1))/2 \times n \times W\_VEC$. The total number of wires in the semi-1D design is less than the traditional design number. Using the

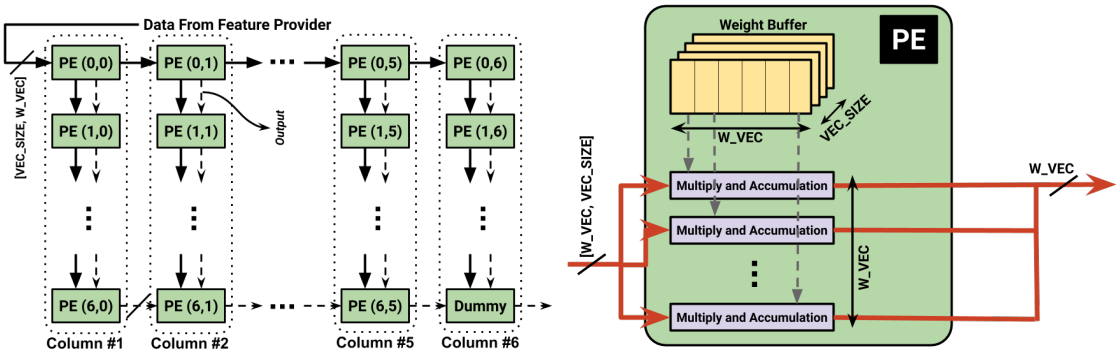
semi-1D arrangement for our design (32 PEs), compared to the traditional design in DLA Aydonat *et al.* (2017), we can reduce the total number of output ports by 65%. Respectively, it reduces the total Flip-Flops (FFs) consumption by up to 25%.

The number of PEs (P) can affect the area efficiency of the semi-1D architecture. For example, for a design with 29 PEs, we cannot have a perfect grid with exactly 29 PEs. The best grid will be an 8×4 grid, with 3 PEs that only work as bypassing data. This configuration would consume more resources than necessary, and can lead into inefficiect utilization of the available resources on the chip. We need to mention that this problem can be alleviated by configuring the extra PEs to take care of the next input data set. This approach requires additional scheduling and management from the controller.

In another design architecture, we eliminated the output channels and instead used data channels to transfer the outputs. This design is not fully pipelined since data and output are sharing the same channel, and the data feeding process stalls while PEs are generating the output. This optimization increases the execution time by 4% for C3D and VGG-16 while reduces total FF consumption by 32%. Since our framework is ultimately focused on the overall performance, we prefer the previous design architecture.

Figure 5.3.b depicts the overall architecture of the PEs. Each PE has W_VEC number of multiply-accumulators (MACs). Each MAC receives an array of data with the length of *VEC_SIZE*. Further, it fetches the respective array of weights of the same size from the buffer and performs an element-wise MAC between these two arrays. The output of all MACs (an array of size W_VEC) is added and stored into a local buffer of the same size, which acts as a temporary buffer for the partial accumulation. To fully process a convolution, each PEs iterates the above process for $Height \times (IN\_CH\_SIZE \div VEC\_SIZE)$ number of times. After fully iterating a

single convolution, the PE streams the content of the intermediate results buffer to the output channel.



(a) (b)

Figure 5.3: (a) Processing Element Semi-1D Structure, (b) Processing Element Architecture

Fully-Connected (FC) PE CNNs usually consist of convolutional FC layers and simple FC layers. Convolutional FCs are the layers that connect the earlier convolution layers to the very first FC layers and have higher memory intensity. For example, the single convolutional FC layer in VGG-16 (Layer #13) has at least 4.3 times higher number of parameters, compared to all other layers. Simple FC layers receive the input from a previous FC layer and perform matrix multiplication. Simple FCs have higher data to computation ratio, compared to the convolutions. Such a difference requires designers to optimize the FC operations on a target hardware architecture. Prior CPU and GPU implementations use the regular FC representation to utilize available libraries, such as MKL on Intel CPUs and cuBLAS on Nvidia GPUs. Unfortunately, using regular FC representation for convolutional FC layers can impact the performance and introduce significant data duplication overhead, which can overwhelm the FPGA bandwidth-limited DDR. Zhang et al. Zhang *et al.* (2018) showed around 25 times overhead for using the above approach.

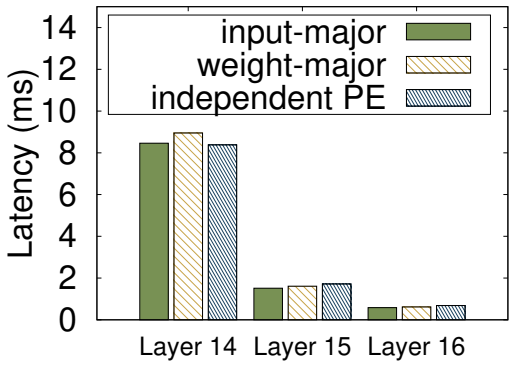


Figure 5.4: Performance for Different Mappings of the VGG-16 MM Pperations.

The optimal acceleration of FC layers requires efficient mapping of these layers onto the FPGA hardware. A common approach is to map the FC matrix multiplication onto the available systolic array. To map the FC layer onto the systolic array, the user has two options: (1) input-major, and (2) weight-major mappings. Zhang et al. Zhang *et al.* (2018) studied the efficiency of these mappings onto their 2D systolic array architecture for a batch of input data. Since our design performs inferences on a single input at a time, the previous observations may not be applicable anymore. Also, the architectural differences between the two designs can affect the choice of mapping. As a result, we experimented with both input-major and weight-major mappings for a set of FC layer configurations. For the input-major mapping, the input channel dimension is divided by W_VEC, where W_VEC number of arrays of the size VEC_SIZE are mapped onto each plate of the *Feature Provider*. Doing so enables efficient utilization of the computation capacity that exists in similar convolution layers and reduces the total number of iterations required for the output calculation. For the weight-major mapping, each *PE* fully loads the input data (instead of the weights), and the *Feature Provider* feeds the weight parameters into the *PE* array, in the same fashion. Doing so enables higher data-reuse rate for the input data, and reduces the load from the external memory.

Both the input-major and weight-major mappings' performance are bounded by the global memory bandwidth. In contrast to convolutions, in *Matrix Multiplication (MM)* operations, the Weight Provider needs to update the PEs in both input-major and weight-major mappings frequently. As a result, for single input inference, the total latency of the MM is bounded by the memory bandwidth. Figure 5.4 represents the latency of matrix multiplications for the last three layers of the VGG-16 model. Based on these results, both weight-major and input-major mappings provide the same latency.

Unlike all previous related works, we propose the independent MM acceleration approach. This independent PE is responsible to accelerate the matrix-multiplication operations, without blocking the rest of the systolic array. As we mentioned above, the computation of the fully-connected layers is bounded by the total global memory bandwidth. As a result, the common strategy (used by all previous related works) of accelerating matrix multiplications on the available systolic array cannot fully exploit the parallelism of the PE systolic array. Table 5.1 represents the experimental performance evaluation of various FC layers and the theoretical maximum performance, considering the number of parameters and the effective bandwidth of the global memory (12 GBps). The input and output sizes are based on the typical FC layer dimensions in practical neural networks. The results confirm that using all the available PEs cannot guarantee performance improvements, compared to the theoretical performance cap. We need to mention that, on average, all our results are 0.25ms slower than the theoretical maximum performance. This difference is due to the overhead of the whole design, which includes the startup time of the OpenCL stack, on both the host and the device. We expanded our experiments to find the essential number of PEs to efficiently accelerate the fully-connected layers and avoid wasting extra computing resources. We modified the total number of PEs and evaluated the

input/output	256	512	1024	2048	4096
	32-PE — 1-PE — Theor.				
2048	0.27ms — 0.34ms — 0.04ms	0.26—0.39—0.08	0.42—0.46—0.17	0.58—0.65—0.33	0.94—1.06—0.67
4096	0.3—0.4—0.08	0.36—0.47—0.17	0.58—0.69—0.33	0.93—0.95—0.67	1.51—1.63—1.33
8192	0.43—0.54—0.17	0.51—0.65—0.33	0.94—0.96—0.67	1.56—1.63—1.33	2.89—2.94—2.67
16384	0.53—0.64—0.33	0.86—0.94—0.67	1.52—1.6—1.33	2.94—2.91—2.67	5.57—5.65—5.33
25088	0.77—0.78—0.51	1.24—1.42—1.02	2.27—2.29—2.04	4.36—4.3—4.08	8.46—8.3—8.12
32768	0.9—0.94—0.67	1.56—1.64—1.33	2.86—2.98—2.66	5.57—5.49—5.28	10.91—10.75—10.62

Table 5.1: Performance Comparison between 32-PE, 1-PE, and Theoretical for Acceleration of the Fully-connected Layers.

performance of the design. Our results confirm (shown in Table 5.1) that a single PE is sufficient to saturate the memory bandwidth thoroughly, and can provide the same performance as 32 PEs.

Using a separate PE for fully-connected layers also enables our accelerator to co-locate the processing of the last fully-connected layers of one input and the first convolutions of the next input in time-shared scenarios. As a result, the FPGA pipeline will not be blocked by the matrix multiplication, which increases the delay for receiving new inputs for processing. For example, the new design reduces the interval of receiving new input requests for VGG-16, from 30.02ms to 26.52ms.

Winograd Transformers Winograd transformation technique is used to increase the number of operations per clock cycle. Our design utilizes this technique by transforming the input feature data into the Winograd representation. In this design, we set the length of the Winograd data in the x-axis to eight. This stage is handled by the *Winograd Transformer*. The generated data is fed to the array of PEs for processing. The output from the final PE should be inversely transformed to represent the real final value. This stage is handled by the *Winograd Inverse Transformer*. For example, in our design, the output always has a constant length of eight, in the x-axis dimension. For the convolutions of sizes three and seven, which are the typical convolution dimension, the size of the output after the inverse transformation

is six and two. We need to mention that we do not apply any transformations on the weights since they are already preprocessed and transformed into the Winograd representation on the host for any number of inferences.

(De)Serializer Our design utilizes a specific *Deserializer* and *Serializer* to receive/send data from the previous or to the next FPGA. For deserialization, the *Deserializer* receives the size of the incoming data from the controller, and then starts receiving data in *long4* format, which is an array of four *long* values. Further, it stores the received data in a specific global memory buffer. Similarly, for serialization, the *Serializer* receives the size of the data to be sent and the location of the data from the controller. Afterwards, it starts sending the data in that buffer to the next FPGA, again in *long4* data format.

xBar Interconnect Our design adopts the traditional *xBar* in DLA Aydonat *et al.* (2017). *xBar* is a custom interconnect to customize, connect, and configure the layers to the design. This component allows for adding more layers and achieves higher acceleration. Examples of these layers are ReLU, Pool, Norm, and Sigmoid.

5.1.2 3D CNN Accelerator Architecture

3D CNNs are composed of 3D convolution layers, which have higher computational intensity compared to the 2D convolutions. This higher intensity is due to the existence of an extra dimension (usually frames) in the input feature map, the intermediate feature map between the layers, and the weight filters. 3D CNNs rely on this extra dimension to learn the motion information across multiple frames of a video. Toward this goal, 3D convolution weights slide over the frame dimension on the input feature map, similar to the weights sliding over the width and height to generate the convolved output. The semi-1D CNN accelerator can be extended to support 3D convolutions. Unfortunately, mapping the 3D convolutions onto the semi-

1D systolic array is not trivial and can lead to poor performance or low utilization of the FPGA resources. There are two main challenges in mapping the 3D convolutions onto the systolic array. First, The choice of order between processing the frames or the output channels can lead to different data and weight re-use, which can directly affect the performance. Second, the large weight size of the 3D convolutions may not fit inside the limited local buffer of the PEs. In the following, we discuss our solutions to address these two problems.

Proper order between dimensions: Choosing the right processing order between the frame dimension and the output channel dimension can affect the frequency of global memory accesses and the data re-use rate, and is thus critical for achieving the fastest implementation. There are two approaches for mapping the computation of the 3D convolution onto the systolic array. In the first approach, the weight provider uploads the weights into the PEs for a certain number of output channels (32). Further, the feature provider can slide over the frames of the input data and stream the data to the PE array. At the end of this round, we have all the output frames for that specific number of output channels (32). The accelerator performs this process for "OUT_CH_SIZE / NUM_PE" rounds, where NUM_PE equals to 32, to get all the relevant data for a set of output channels. In this approach, each round, the accelerator generates all the output frames for a set of output channels, which we call *Output-Major* approach. In the second approach, the feature provider caches a frame of data in the local buffer and streams it into the PE array. The weight provider iteratively loads the weights into the PEs. Further, PEs convolve the input data with the weights and generate the outputs. For the next round, the feature provider slides one frame over the input data and starts over the same process. In this approach, each round the accelerator generates all the output features for one output frame, which we call the *Frame-Major* approach.

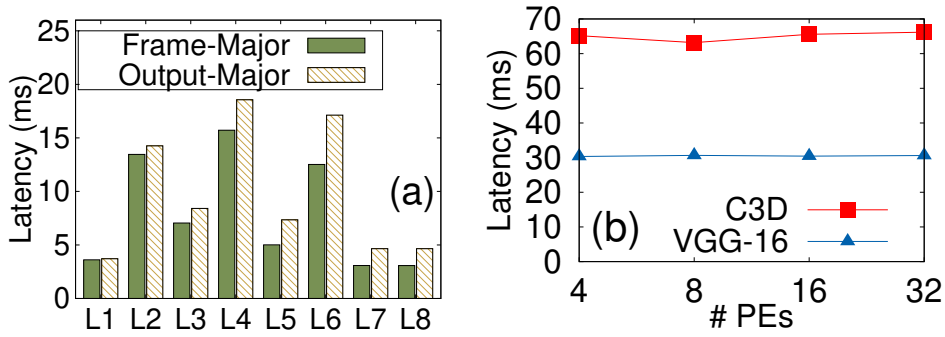


Figure 5.5: (a) Latency of the Layers of C3D Model, with *Frame-Major* and *Output-Major* Approaches, (b) Performance of C3D and VGG-16, with Different Number of PEs. It's Already Included in the Graph.

Figure 5.5a represents the latency of the layers of a sample 3D CNN model (C3D) while experimenting with the above two approaches. In comparison, the frame-major approach outperforms the output-major, up to 1.3 times faster for specific layers. As a result, our design adopts the frame-major approach for 3D convolutions.

Large number of weight parameters: Unlike 2D convolutions, 3D convolutions have a large number of weight parameters, due to the existence of the extra frame dimension. As a result, the weight size might exceed the available local buffer in the PEs. There are three approaches for addressing this problem. First, we can split the weight features into *NUM_CHILDLAYER* child layers, where each child layer handles a portion of the weights, which can entirely fit in the PE's local buffer. The same approach is also adopted by Liu et al. Liu *et al.* (2019). The weights can be split from either the input-channel or the frame dimension, where both lead to the same performance. The outputs of all the child layers have to be accumulated to get the final output of the convolution for the whole weight features. This accumulation is performed inside the *Feature Writer*, which writes back the data to the global memory (Figure 6.5). Second, we can trade the number of PEs with the total amount

of memory available in each PE. By reducing the number of PEs, we can allocate a larger buffer for each PE, which can store the whole weight parameters. Third, we can use a combination of the above two approaches. Considering the *"weight splitting"* and *"fewer PEs"* as the two ends of the spectrum, it is possible to have an architecture where sits somewhere in between. For example, we can attempt for reducing the total number of splits, while maintaining a minimum number of PEs.

We evaluated the above approaches by experimenting with various designs with different numbers of PEs and buffer size. We synthetically reduced the total size of the local memory on the FPGA to make the PE buffer small for the weight parameters. Figure 5.5b represents the performance of the VGG-16 and C3D models while running on the FPGA with different configurations (number of PEs). On the two ends of the spectrum, we have the 32 PE configuration (maximum number of PEs) and the 4 PE configuration (enough buffer to hold all the weight parameters). We also explored architectures with 8 and 16 PEs, which represent the trade-off between the number of PEs and the available buffer on each PE. Need to mention, reducing the number of PEs to increase the buffer size does not mean the allocation of less DSPs. In this configuration, we can assign more DSPs to each PE by increasing the value of VEC_SIZE in our design (each PE contains $VEC\_SIZE \times W\_VEC$ number of DSPs). As a result, we can maintain the same level of parallelization.

Experimental results show almost no difference between various configurations. As a result, we decided to stick with the default configuration (32 PEs, with smaller VEC_SIZE value), which requires weight splitting for layers with a large number of weight parameters. Increasing the value of VEC_SIZE can have adverse effects for layers with a small number of channels. In these cases, VEC_SIZE goes beyond the size of the channel, which leads to having some of the DSPs going idle, or doing dummy computations. As a result, choosing a relatively small VEC_SIZE value can

make the design generic enough for all types of layers with different configurations, irrespective of their size.

5.1.3 Multi-FPGA Support

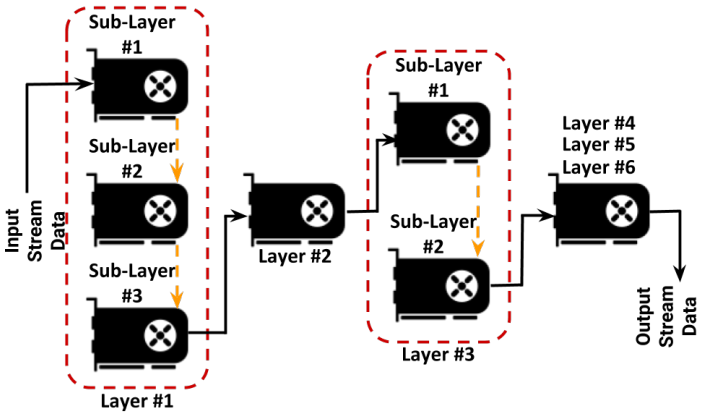


Figure 5.6: Mapping a Neural Network Onto a Cluster of FPGAs.

Our framework can split a single CNN design onto multiple FPGAs, connected through Intel-supported serial channels. Each serial channel utilizes a 40Gbps Infiniband link. The FPGAs are connected in a daisy-chained fashion. The first port of each FPGA is connected to the second port of the previous FPGA in the chain, receiving the data. The second port of each FPGA is connected to the first port of the next FPGA in the chain, sending the data. The first port of the first FPGA and the second port of the last FPGA do not have any cables connected. Need to mention, the first FPGA can be connected to a streaming device (camera, cloud, etc.), and directly receives the data for processing.

The assumption for achieving linear speedup is the perfect load-balance of the FPGAs, while running the layers. In another words, each FPGA should handle an equal computation load of the whole execution. An imperfect load-balance leads to one or more FPGAs handling a larger share of computation, compared to the other

FPGAs. Those FPGAs will become the bottleneck in the pipeline, and stops the design to achieve perfect linear speedup. As a result, proper load-balance is critical to enable optimal throughput improvement.

Figure 5.6 depicts a sample CNN deployed on a series of FPGAs in a row. In this example, we broke large layers into multiple smaller sub-layers. We also grouped multiple small layers, where the granularity of the group is equal to the other layers or sub-layers. This approach enables our design to balance the computation overhead among all FPGAs. During the execution, each FPGA receives the input or the intermediate data from the source or the previous FPGA, pushes the data through the assigned (sub)layers, and sends the output to the next FPGA or the sink. On each FPGA, the sub-layers or the layers are processed sequentially. After processing one input and forwarding the output, each FPGA grabs the next input. In this setup, all FPGAs operate in-parallel to co-locate the execution of different layers of different inputs and increase the overall throughput. In other words, let's consider a network with an execution time of t on a single FPGA. With a configuration of two FPGAs, each FPGA can handle half of the network. In this configuration, the second FPGA can execute the second half of the network for an input, while the first FPGA can start processing the first half of the network for the next input. As a result, we can obtain twice the throughput, compared to a single FPGA configuration.

As mentioned above, our framework breaks large layers into multiple sub-layers. Mapping original layers onto the FPGAs can lead to unbalanced computational overhead throughout the pipeline, and further prevents the framework from achieving linear speedup, while increasing the number of FPGAs. Our framework tackles this problem by splitting a layer from *output* channel into multiple sub-layers with 32 output channels. Any number below 32 is not going to have any benefit since we have 32 PEs that need to finish execution. Also, it breaks a layer over the *frame* channel,

where the number of the frames of the input is higher than the number of frames of the weight. Further splitting forces the design to pad the frame dimension, which introduces unnecessary extra overhead. After the splitting process, each sub-layer can run individually, with no dependency on any other sub-layer.

We need a proper model of our system to find the best mapping of sub-layers onto the FPGAs. We consider our multi-FPGA system is a pipeline of M stages, where M is the number of FPGAs. The framework splits the CNN network into M partitions (each partition is composed of layers and sub-layers) and distributes them sequentially on the chain of FPGAs. We define the *Latency (L)* of the multi-FPGA system, $L_{Multi-FPGA}$, as the time interval, after which the multi-FPGA system can accept the next input. The throughput of our multi-FPGA system is equal to $1/L_{Multi-FPGA}$, which is the total number of inputs it can process per unit of time. As mentioned before, our multi-FPGA system is a pipeline, where each FPGA is called a stage. Each stage handles part of the neural network. In a pipeline, the speed of the whole system is bounded by the speed of the slowest stage (highest latency). Thus, the partition with the highest latency stalls the whole pipeline and is called the bottleneck stage. The latency of the whole pipeline is equal to the latency of the bottleneck stage.

The latency (L) of each layer i can be calculated as follows:

$$\begin{aligned} L_i = (&WEIGHT\_SIZE_i \ / \ DDR\_bandwidth)+ \\ &(Total\_multiplications_i \ / \ (Frequency\times \\ &VEC\_SIZE \times W\_VEC \times Num\_PEs)) \end{aligned} \tag{5.1}$$

In Eq 5.1, the latency of each (sub)layer is specified as the time it takes to read the weights into the PEs, plus the total time it takes to finish the overall calculations. We need to mention that we omit the overhead of the network communication since FP-

GAs are directly connected to each other through 40Gbps QSFP+ Infiniband cables, which makes the communication latency ($OUTPUT\_SIZE/Network\_bandwidth$) negligible. For example, for the C3D model, the maximum communication latency between two FPGAs is less that 0.1ms.

Next, we need to find the appropriate mapping of the (sub)layers onto the FPGAs in our multi-FPGA setting. Assuming we have M FPGAs, connected in a daisy-chain fashion, we aim to map a CNN composed of N (sub)layers onto these FPGAs, in a linear fashion. The most appropriate mapping should lead to the highest possible throughput. The overall throughput in this architecture is typically bounded by the FPGA with the highest latency for handling the assigned layers. We can formulate this problem as balancing the load across the FPGAs in the chain.

To find the most appropriate mapping, we first calculate the processing time (latency) and latency of each (sub)layer, using Eq 5.1. Using these latencies, we can design an algorithm to perform a brute-force enumeration to find the best mapping. For the brute-force enumeration, the algorithm requires to evaluate $\binom{N}{M-1}$ configurations. The time complexity of this method is $O(N^{min(M,N-M)})$, which is exponential to the number of (sub)layers N. We also developed this algorithm in C++ to calculate the best mapping using a different number of layers and FPGAs. For example, a configuration of 100 layers and 10 FPGA takes around 960 hours to finish, which is quite expensive.

To address the above problem, we developed a polynomial-time load-balancing algorithm using dynamic programming. Eq 5.2 presents the overall solution for the optimal mapping of the layers:

$$L_{j,k} = \begin{cases} \sum_{l=1}^{j} L_l & \text{if k = 1} \\ min_{r=1}^{j-1}(max(L_{r,k-1}, \sum_{l=r+1}^{j} L_l)) & \text{if k > 1 and k} \leq \text{M} \end{cases} \tag{5.2}$$

In Eq 5.2 $L_{j,k}$ represents the latency of the first k FPGAs, while handling the (sub)layers from 1 to j. For the case with only one FPGA, the latency for accepting new inputs is equal to the latency of handling all (sub)layers, from 1 to j, on a single FPGA. For the cases with multiple FPGAs, the latency is calculated based one assigning the last few (sub)layers (from $r+1$ to j) in the set to the last FPGA. The latency of the final FPGA equals to the total latency of the layers. Further, the rest of the first (sub)layers (from 1 to r) is mapped to the other FPGAs, which is the sub-problem in our dynamic-programming algorithm. Finally, the latency of the whole pipeline is the latency of the bottleneck stage. The time complexity of this method is $O(M^2 \times N)$, which is linear to the number of (sub)layers. Finally, we can obtain the overall throughput of the system by calculating the latency of the bottleneck stage.

5.2 Experimental Results

To confirm and quantify the benefits of our new framework, we implemented and evaluated *VGG-16*, *AlexNet*, *ResNet*, *C3D*, and *I3D*. *VGG-16* is widely-used for image classification. *ResNet* is a well-known CNN model for object recognition. *C3D* and *I3D* are both video analytics 3D CNNs. We evaluate the performance of our framework in comparison to the available single-FPGA and multi-FPGA solutions. All the experiments were conducted on a set of Intel Fog Reference Design units Intel (2017), each equipped with two Nallatech p385a FPGA acceleration cards. Each host has one Intel Xeon CPU E5-1275, with 32GB of main memory. Each FPGA card has an Intel Arria 10 FPGA, with 8GB of DDR3 SDRAM. FPGAs are serially connected through QSFP+ 40Gb/s InfiniBand cables. The OpenCL kernels were compiled using Intel FPGA SDK for OpenCL (version 19.1) with Nallatech *p385a_sch_ax115* board support packages (BSP). We performed GPU experiments on two Nvidia GPUs: 1) Nvidia RTX 2080Ti, which is a server class GPU, and 2) Nvidia Tesla T4, which is a

small-scale GPU for edge systems. In addition, we performed our CPU experiments on a host, equipped with two Intel Xeon Silver 4210 octa-core CPUs, and 64GB of main memory. We report the latency of our design, which is the time it takes for the design to accept a new input request, following the previous request. Also, we report the throughput as the number of images per second *(img/s)* that the system can accept. The results include the energy-efficiency of our experiments. We used *nvidia-smi* command line utility and the nallatech mempory-mapped device layer API to query instant board-level power consumption. For the CPU, we used the *powerstat* toolkit to measure the power consumption. Finally, we performed post-training quantization with Pytorch on the VGG-16 model to enable it running with a lower bit-precision (8-bit) on the CPU.

In the single-FPGA experiments, we evaluated the performance of our design, for accelerating different CNN models. It achieves state-of-the-art performance on a single FPGA, which is crucial for any further extensions. In the multi-FPGA experiments, we accelerated the same CNN models, but with more than one FPGA. We designed the experiment to show the correlation between the number of FPGAs and throughput.

5.2.1 Single-FPGA Performance Evaluation

In this chapter, we present the effectiveness and performance of our design on a single FPGA. Having a state-of-the-art single-FPGA solution is the basis for extending that solution to multiple FPGAs. Tables 5.2 and 5.3 present the performance, energy-efficiency, and resource utilization of our design and the related works [1] . Our solution can provide 27ms of latency for single input inference on VGG-16, which is

[1]Because the design details and source code are often not available, comparing to the performance numbers reported in the related papers is the standard practice in the FPGA community.

faster than all other related works Ma *et al.* (2018); Suda *et al.* (2016); Zhang *et al.* (2018); Wang *et al.* (2017). For ResNet-50, which is a widely-used object detection model, our solution can achieve 21ms of latency per single input, which is 6ms faster than Ma et al.'s solution Ma *et al.* (2017b).

DLA Aydonat *et al.* (2017) can achieve an average of 1ms latency for each image, while processing a batch of images. This low latency is achieved by reading the weight and input data from the local memory, which eliminates the overhead of accessing global memory. We replicated the DLA experiment with VGG-16 and achieved 37ms of latency per single image. By offloading the FC layers onto a dedicated PE, we can achieve 27ms of latency, which is 10ms faster than our clone of DLA.

Our approach achieves similar or better energy-efficiency than the related works, except the AlexNet on the Intel DLA. In this specific example, the design avoids reading/writing from/to the external memory (AlexNet model can fit inside the on-chip memory) and applies inference on a batch of data. External memory exclusion significantly reduces the power consumption, and the utilization of a batch instead of a single input increases the matrix multiplication operation's performance. On the other hand, our design targets streaming applications, requiring real-time service of each input.

We also demonstrate the performance of our framework for accelerating CNN models with 3D convolutions. Table 5.4 presents the performance of some standard video processing CNNs, from our FPGA solution and the performance reported by the related works. Using our design for 3D convolutions, we can achieve almost 1.7 times better latency compared to these related works. This lower latency enables processing of a higher number of frames per second, for latency-critical applications.

Finally, Table 5.5 presents the performance and energy-efficiecy of our design, compared to the state-of-the-art CPU and GPU implementations. The CPU shows a

high latency, while handling a single input for inference, which makes it inferior to the accelerators. It also results in the lowest energy-efficiecy. The GPU can outperform the FPGA, but the energy-efficiency of the FPGA is much better. Need to mention, the FPGA can operate individually, while the GPU requires a host, which adds to the overall power consumption. In addition, FPGAs can ingest data directly into the pipeline, which is useful to many stream data processing scenarios such as edge computing Biookaghazadeh *et al.* (2018), whereas GPUs require the data to be first stored in the host memory and then copied to the GPU memory.

5.2.2 Multi-FPGA Performance Evaluation

In this chapter, we evaluate the performance of our framework, while scaling-out the CNN deployment on multiple FPGAs. We performed all experiments on a pipeline of FPGAs (from one to eight) connected through I/O channels. We evaluated the performance of both 2D and 3D CNNs on our multi-FPGA platform, using VGG-16, C3D, and I3D. Figure 5.7 presents the performance improvements while accelerating the three CNN models mentioned above, using multiple FPGAs. For all three models, the throughput increases linearly by increasing the number of FPGAs, from one to eight. We can observe that using the load-balancer in Chapter 5.1.3 can lead to a near-perfect partitioning scheme with a fully balanced workload across the FPGA.

We also made a comparison between our solution and the other available multi-FPGA related works Jiang *et al.* (2019); Zhang *et al.* (2016). In comparison, our solution provides up to 5.8 times better throughput. Jiang et al. Jiang *et al.* (2019) experimented using 16-bit variables, which has higher precision than our configuration (8-bit). They can achieve similar performance to ours by utilizing a lower-precision configuration, but that they are using a better FPGA (much more resources, compared to Arria 10). Their solution cannot scale linearly beyond 4 FPGAs for the

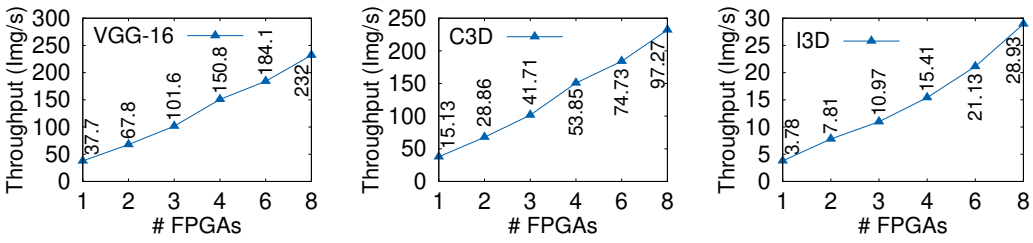


Figure 5.7: Acceleration of 2D and 3D CNN Models Using Multiple FPGAs.

VGG-16 model. Also, our design supports a diverse set of convolutional operations, and it is based on the high-level OpenCL language. Finally, our solution achieves much higher energy-efficiency. Need to mention, the power consumption of the related work Jiang *et al.* (2019) for four and eight FPGAs was estimated since the work provided experiments for only two FPGAs and simulated the results of more than two FPGAs.

5.3 Conclusions

In this work, we demonstrated a high-throughput multi-FPGA acceleration solution for a wide variety of CNNs. We first extended the DLA Aydonat *et al.* (2017) architecture to achieve lower latency and better resource utilization, for single FPGA configuration. Second, we extended the architecture to support 3D convolutions for the video understanding applications. It includes studying the optimal deployment of the 3D convolutions on the semi-1D systolic array. Third, we enabled communication between the FPGAs to support multi-FPGA setups. Finally, we developed an algorithm for automatic deployment of the CNN layers onto the FPGAs in the multi-FPGA setup. Our results show that utilizing multiple FPGAs can linearly increase the overall throughput of the CNN inference. Also, optimizing the PE systolic array and FC operations can lead to better area utilization (up to 25%) and higher overall throughput (up to 24%), compared to the state-of-the-art CNN accelerator. Finally,

we studied the efficient mapping of 3D convolutions on our novel semi-1D systolic array to achieve the highest overall throughput.

Design	Zhang	Ma	Ma	Suda	Zhang		Wang	Aydonat		Ours			
CNN Model	Alex Net	VGG -16	Res Net -5	VGG -16	VGG -16		VGG -16	Alex Net	VGG -16	Alex Net	Res Net -50	VGG-16	
FPGA	Virtex 480t	Arria 10	Arria 10	Stratix V	KU060		Arria 10	Arria 10	Arria 10	Arria 10			
Clock Frq (MHz)	100	200	150	120	200		190	303	215	212			
Precision (bits)	32	16	16	16	16	8	8	8	8	8		8	16
Latency/ Image (ms)	43.23	43.2	27.2	117.8	101.15	25.3	225	1	37	8.8	20.9	26.52	30.3
Through put (GOPS)	61.62	715.9	285.07	262.9	266	1.17K	N/A	1300	N/A	990		990	866.25
Through put (Img/s)	23.13	23.14	36.7	8.48	9.88	39.52	13.45	1000	27	113.6	47.8	37.7	32.9
Power (watt)	18.61	N/A	N/A	N/A	25		27.3	45		23			

Table 5.2: Performance Comparison of State-of-the-art Single-FPGA Implementations. Each Column Represents One of the Related Works, Including Ours. Each Row Represents Some of the Configurations of the Experiments, and the Performance and Resource Utilization of the Related Works. Baselines are Zhang *et al.* (2015), Ma *et al.* (2018), Ma *et al.* (2017b), Suda *et al.* (2016), Zhang *et al.* (2018), Wang *et al.* (2017), Aydonat *et al.* (2017)

Design	Zhang	Ma	Ma	Suda	Zhang		Wang	Aydonat		Ours		
CNN Model	Alex Net	VGG -16	Res Net -5	VGG -16	VGG -16		VGG -16	Alex Net	VGG -16	Alex Net	Res Net -50	VGG-16
FPGA	Virtex 480t	Arria 10	Arria 10	Stratix V	KU060		Arria 10	Arria 10	Arria 10	Arria 10		
Clock Frq (MHz)	100	200	150	120	200		190	303	215	212		
Precision (bits)	32	16	16	16	16	8	8	8	8	8	8	16
DSP (Used / Total)	2.2K / 2.8K	1.5K / 1.5K	1K / 1.5K	0.8K / 1.9K	1K / 2.7K	0.1K / 2.7K	0.5K / 1.5K	1.5K / 1.5K	1.5K / 1.5K	1.4K / 1.5K	1.4K / 1.5K	1.2K / 1.5K
BRAM (Used / Total)	1K / 2K	1.5K / 2.7K	2.1K / 2.7K	1.6K / 2.5K	0.7K / 2.1K	0.7K / 2.1K	N/A	2.4K / 2.7K	1K / 2.7K	1.3K / 2.7K	1.3K / 2.7K	1.7K / 2.7K
LUT (Used / Total)	186K / 303K	N/A	N/A	N/A	100K / 320K	200K / 320K	N/A	N/A	278K / 854K	290K / 854K	290K / 854K	420K / 854K
FF (Used / Total)	205K / 607K	N/A	N/A	N/A	80K / 727K	140K / 700K	N/A	N/A	725K / 1708K	762K / 1708K	762K / 1708K	933K / 1708K

Table 5.3: Resource Utilization Comparison of State-of-the-art Single-FPGA Implementations. Each Column Represents One of the Related Works, Including Ours. Each Row Represents Some of the Configurations of the Experiments, and the Performance and Resource Utilization of the Related Works. Baselines are Zhang *et al.* (2015), Ma *et al.* (2018), Ma *et al.* (2017b), Suda *et al.* (2016), Zhang *et al.* (2018), Wang *et al.* (2017), Aydonat *et al.* (2017)

Design	Shen Shen *et al.* (2018)	Liu Liu *et al.* (2019)	Ours
CNN Model	C3D	C3D	C3D
FPGA	VC709	VC709	Arria 10
Clock Frq (MHz)	N/A	120	210
Precision	N/A	N/A	8-bit
Latency (ms)	89.4	115.5	66.08
Throughput (GOPS)	427.5	667.7	990
Throughput (Input/s)	11.18	8.65	15.13
Power (watt)	25	25	23
Energy Efficiency (GOPS/watt)	17.1	26.7	43.04
Energy Efficiency (Input/J)	0.44	0.34	0.65
DSP Util.	1.5K / 3.6K	3.5K / 3.6K	1.5K / 1.5K
BRAM Util	1.5K / 2.9K	0.3K / 1.4K	1.3K / 2.7K
LUT Util.	242K / 432K	272K / 432K	290K / 854K
FF Util.	286K / 866K	434K / 866K	762K / 1708K

Table 5.4: Performance Comparison of Single-FPGA Video Processing CNN Acceleration. Columns and Rows are the Same as Table 5.2.

Accelerator	RTX 2080Ti	Tesla T4	Xeon CPU		Ours	
Clock Frq (MHz)	1545	1087	2200		212	
Precision (bits)	32	32	32	8	8	16
Latency/Image (ms)	8.43	13.14	128.39	58.35	26.52	30.3
Throughput (Img/s)	118.6	76.10	7.78	17.13	37.7	32.98
Power (watt)	250	70	86		23	
Energy Efficiency (Image/J)	0.47	1.08	0.09	0.19	1.6	1.4

Table 5.5: Performance Comparison Between CPU, GPU, and Our FPGA Implementation, Running the VGG-16 Model.

Design	Zhang		Jiang				Ours			
CNN Model	VGG-16		VGG-16				VGG-16			
FPGA	VC709		ZCU102				Arria 10			
Clock Frq (MHz)	150		200				210			
Precision	16-bit		16-bit				8-bit			
Num. FPGAs	1	2	1	2	4	8	1	2	4	8
Throughput (Input/s)	6.5	13	14	35.28	74.2	84	37.7	67.8	150.8	232
Throughput (GOPS)	203.9	407.8	N/A	N/A	N/A	N/A	990	1980	3960	7920
Power (watt)	25	50	27.2	54.4	108.8	217.6	23	46	92	184
Energy Efficiency (GOPS/watt)	8.16		N/A				43			
Energy Efficiency (Input/J)	0.26		N/A				1.63			

Table 5.6: Performance Comparison of Multi-FPGA Acceleration Solutions. Baselines are Zhang *et al.* (2016), and Jiang *et al.* (2019)

Chapter 6

THROUGHPUT-AWARE SCHEDULING OF RANDOMLY-WIRED NEURAL NETWORKS ON MULTI-FPGA PLATFORMS

Per our introduction in chapter 1.4, the current scheduling methods for the available multi-FPGA setups are not designed for RWNNs, due to their complicated connectivity. As a result, RWNNs cannot fully benefit from running on these multi-FPGA accelerators. In this chapter we propose a novel scheduler that efficiently splits and maps the layers of an RWNN on the FPGAs and enables throughput maximization on the pipeline. We also discuss our solutions for the rising challenges, scheduling RWNNs on the multi-FPGA setups.

6.1 Challenges

The scheduling methods in current compilers Chen *et al.* (2018); Vasilache *et al.* (2018), frameworks Abadi *et al.* (2016); Paszke *et al.* (2019); Jia *et al.* (2019), and related works are designed for conventional neural networks, where the network embodies a topological order amongst the operations. Such limitation makes it quite challenging for these schedulers to optimally schedule RWNNs on the streaming processing architectures, such as multi-FPGA systems. Currently, these schedulers tackle this problem by considering a random topological order of the operators, which may lead to non-optimal scheduling. In this chapter, we discuss these challenges in-depth:

Task Placement. Proper neural network operator placement is critical to achieve maximum available throughout, or the minimum possible latency. An optimal task placement usually minimizes the difference of loads (typically execution load) amongst the available processors. A large load difference between two processors makes the

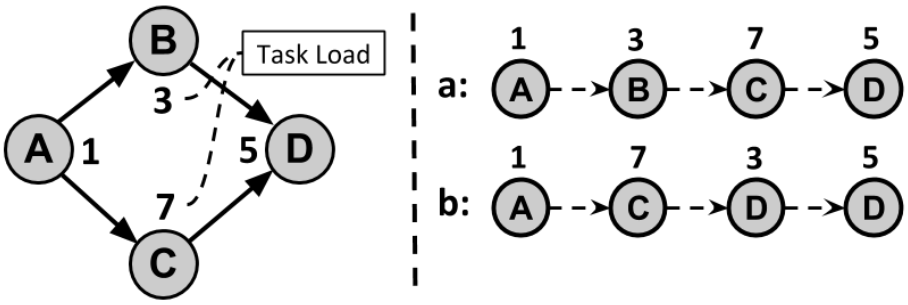


Figure 6.1: The Figure on the Left is a Simple Graph of Four Operations. A Dependency of Task B on Task A is Represented as $A \rightarrow B$. Each Task is Assigned with a Load Value. On the Right, We Have Two Possible Topological Sorts of the Same Graph. Option (a) Leads Into a Non-balanced Task Distribution, Regardless of the Distribution. Option (b) Can Lead to Perfect Balance by Placing (A,C) on One FPGA and (B,D) on Another FPGA.

processor with smaller load experience a long idle time. Figure 6.1 presents a set of tasks, where each task carries a certain computation load, and tasks may have data dependency amongst themselves. In addition, it depicts two different topological orderings, where each leads a different distribution. All distributions from the ordering in Figure 6.1.(a) lead non-optimal task distribution, where in the best case (A, B) with a total weight of 4 are located on the first FPGA, and (C, D) on the second FPGA with a total weight of 12. On the other hand, Figure 6.1.(b) contains a completely balanced distribution, where placing A, C and B, D on the first and second FPGAs makes both of them carrying a weight of 8. As a conclusion, available techniques may lead into non-optimal task placement.

Memory Management. The outcome of scheduling places part of the neural network on each FPGA. Each part is a graph of operations, which may embody the same randomness through the connectivity between the operations. The execution order of these operations on a single processor can lead different memory consumption. This

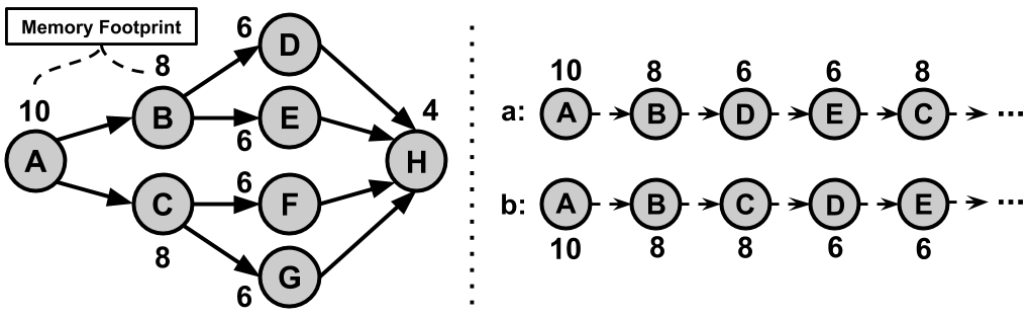


Figure 6.2: On the Left is a Sample Graph with Eight Operations. The Numbers on Each Operation Represent the Unit of Memory Footprint for the Output of that Operation. On the Right is Two Different Scheduling Order of the Operations.

can be a problem for devices with limited memory, where the execution order creates an out-of-memory situation.

Figure 6.2 presents a sample RWNN, with each operation having a different memory footprint. Scheduling of the operations may result in various memory footprints. For example, the scheduling in Figure 6.2.a requires 30 units of memory (to keep the intermediate results for the next operations), while the scheduling in Figure 6.2.b only requires 26 units. As a conclusion, memory is critical for processors with limited storage resources.

In summary, all above challenges are unique to RWNNs, due to random connectivity that can exist between the operations in the graph. Unfortunately, our proposed technique in the previous chapter cannot address the above challenges, since it is mainly designed to address neural networks with linear connectivity.

6.2 Design Objectives

Scheduling Algorithm. We propose a method to find the optimal scheduling S_{opt} from all possible schedules (S) that maximizes the throughput (λ) of the multi-FPGA pipeline, through balanced task scheduling. Each schedule in S is an enumeration of

mapping the operations onto the FPGAs, and the order of execution of each operator $v \in V$ on an FPGA, where V is the set of all nodes in the neural network graph G.

$$S_{opt} = argmax_s \lambda(s, G), for s \in S \tag{6.1}$$

The optimal scheduling problem can be modeled as a graph partitioning problem, where partitions should contain no more or less than a specific number of vertices, which are operators in the neural network. Through balanced graph partitioning we can ensure balanced distribution of operators onto the FPGAs, and reduce the latency of the FPGA with the largest load, hence the idle time of the whole pipeline.

One straightforward way to find the the best mapping is the *brute-force* approach. Unfortunately, the complexity of this approach is $O(|V!|\times|V!|)$, which makes it non-practical and too costly to find the optimal solution. In addition, a number of related works have shown that optimal graph partitioning is an NP-Complete problem, and proposed various heuristics to solve the problem, such as Kernighan-Lin (KL) Kernighan and Lin (1970), Fiduccia-Mattheyses (FM) Fiduccia and Mattheyses (1982), and multi-level partitioning Hendrickson and Leland (1995); Karypis and Kumar (1995). These heuristics can only address non-directed graphs and hypergraphs, while we are focused on Directed Acyclic Graphs (DAG). More recent works have addressed DAG graphs Moreira *et al.* (2017, 2020), but they are reducing the the total edge weights between the partitions, while we need better balancing of the partition weights and not the weights between the edges.

To this end, we explore a modified version of the partitioning algorithms, where it can derive the optimal scheduling S_{Opt} and maintain the acyclicity of the task assignment on the FPGAs.

Memory Optimization The choice of execution order of the operations on a single FPGA can affect the maximum memory consumption ($\mathcal{M}$) on the device. Throughout

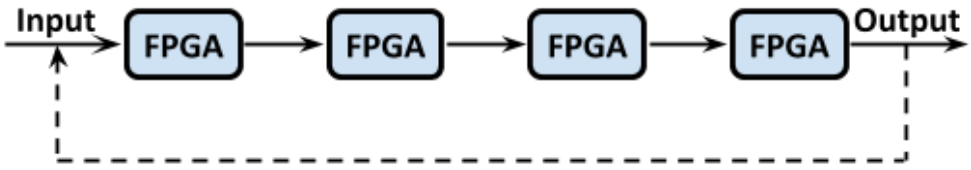


Figure 6.3: Ring- or Chain-style Multi-FPGA Configuration.

the execution, the intermediate outputs need to be held in the memory for further consumption. Different orders of executions affect the lifetime of these intermediate results, which can be accumulated with other intermediate results and increase the memory footprint.

The goal of scheduling on each FPGA is to determine the best execution order ($\mathcal{S}_{opt}^{local}$) of the tasks memory consumption:

$$\mathcal{S}_{opt}^{local} = argmin_s \mathcal{M}(s, G^{local}), for s \in S^{local} \tag{6.2}$$

Our scheduler needs to effectively find the optimal schedule s, from all possible schedules S^{local} of the assigned operators G^{local} on a single FPGA, and pick the schedule with minimum memory footprint ($\mathcal{M}_{min}$).

Operation Division The absence of balance in computational intensity amongst the operations of a neural network makes the optimal schedule ineffective to achieve near-perfect load distribution. For example, in case of I3D Wang *et al.* (2019b), which is a 3D convolutional neural network for human hands gesture detection, the first layer alone consumes more than half of the total inference execution time. In this case, the model cannot scale beyond two FPGAs, irrespective of the scheduling. Similarly, The gap of computational intensity in RWNNs can stop the design to scale beyond a certain number of FPGAs, and introduce unnecessary idle times.

To address this issue, we propose the *Operation Division*, which is rooted from

graph transformation techniques Rozenberg (1997). Our methods take advantage of distributive, associative and commutative properties of the neural network operations, which maintains the semantics of the operations, while enables improvements in the quality of balancing the load amongst the FPGAs. For example, a matrix-vector multiplication of $A \times B = C$, where A is a matrix and B is a vectors, can be re-written as 6.3, where A_1 and A_2 present the first and second halves of the A matrix. As a result, the computation can be divided into two parallel computations, each carrying around half of the execution time.

$$[A_1 \times B, A_2 \times B] = C \tag{6.3}$$

The operation division approach is applicable on almost all neural network operations, such as matrix multiplications, and convolutions. Doing so enables splitting large operations into finer-grain elements, and consequently a more effective schedule.

Scheduling-Aware Inter-FPGA Communication. In an *RWNN*, each node can send or receive data from other nodes with any arbitrary distance in the graph. This is in contrast with traditional networks with chain-wise connection and deterministic communication pattern. Also, most multi-FPGA platforms a ring- or chain-style configuration, where each FPGA is directly connected to the previous/next FPGA in the pipeline (Figure 6.3). Mapping *RWNNs* onto this multi-FPGA configuration requires the platform to have knowledge of the network architecture and proper routing mechanisms for consumption or redirection of incoming data in the pipeline, since not all incoming inputs to an FPGA should be consumed, but rather be transferred down the pipeline.

6.3 Piper: Throughput-And-Memory-Aware Scheduling of RWNNs

Our scheduler (*Piper*) aims to maximize the throughput of RWNNs on multi-FPGA configurations and reduce the memory footprint on every FPGA. Piper leverages a heuristic graph partitioning algorithm to find an optimal partitioning scheme for the DAG of a neural network. Also, it relies on a dynamic programming approach to find the best local schedule of the operation on each FPGA, reducing the memory footprint. Finally, Piper performs operation division, which results in a finer-grained schedule of the operations. In this chapter , we dive deeper into the details of the Piper design.

6.3.1 Scheduling Algorithm

The first goal of Piper is to distribute the RWNN operations onto the FPGAs in a balanced fashion, and each FPGA receives a fairly equal amount of load. To do so, Piper reduces the question into a graph partitioning problem and uses a heuristic to split the graph into K partitions. Here we discuss two heuristics from related work Moreira *et al.* (2017), and later evaluate them in chapter 6.4.2.

Problem Definition: Let $G = (V, E, N, W)$ be a directed graph. The edge weights are defined as $W : E \rightarrow R^+$, and node weights are $N : V \rightarrow R^+$. The algorithms aims to derive K partitions, i.e. $V = \cup_{i=1}^{K} V_i$ and $\forall_{i,j:i\neq j} V_i \cap V_j = \emptyset$. We define a threshold value as $\mathcal{T}$, which caps the load amount on each partition ($\forall_{1\leq i\leq K} : N(V_i) \leq \mathcal{T}$). The value of $\mathcal{T}$ is defined as $\mathcal{T} = (1+\epsilon)\lceil \frac{N(V)}{K} \rceil$. Previous works have shown that the partitioning problem becomes *NP-Complete*, for $\epsilon < 1$. The smaller ϵ value leads to tighter boundary and better partitioning scheme. Finally, the weight of each partition is defined as the total weight of its vertices and the weight between each two partition is defined as the weight of all directed edges from the source to the

destination partition.

We start with an initial partitioning that confirms the partition size threshold. It also makes sure the partitioning scheme upholds the acyclicity, such that there is no directed edge from V_j to V_i, where $j > i$. The scheduler employs a topological ordering of the graph to do so. In other words, it topologically sorts the graph using the Kahn algorithm Arsham and Kahn (1989). Further, it starts from the tail of the ordered graph and picks the vertices until the total weight of the vertices reaches the value of $\mathcal{T}$. The set of selected vertices forms a single partition. After that, it moves forward to create the next partition in the row. At the end of this algorithm, we will have a set of K partitions, where their weights are below the threshold ($\mathcal{T}$), and the graph of the partitions is also acyclical. After the initial partitioning, we move forward with fine-tuning steps, which are heuristics to improve the partitions' balance.

Differently from previous works Moreira *et al.* (2017, 2020), we consider the gain (Δ) as the distance of the partition weights from the optimal weight, which is $(\sum_{v \in V} v)/|V|$. To this end we mathematically defined the gain function as below:

$$\begin{aligned} AVG &= (\sum_{v \in V} v)/|K| \\ \Delta &= \sum_{i=1}^{K} (N(V_i) - AVG)^2 \end{aligned} \tag{6.4}$$

The goal of the heuristic is to reduce Δ through the fine-tuning process.

In the following sub-sections, we present two effective partitioning schemes to tackle the problem. The tuning techniques are similar to the Fiduccia-Mattheyses Fiduccia and Mattheyses (1982) vertex moves but extended to support a broader move exploration and preserving the acyclicity of the target partitioning.

Simple Moves Heuristic: In this heuristic, the algorithm weighs the possibility

Algorithm 6: Simple and Advanced Moves Heuristics Algorithm. In Case of Simple-heuristics, $Predecessors(v)$ and $Successors(v)$ Will Only Look Into Neighbor Partitions. In Case of Extended-heuristics, They Will Look Into Partitions Beyond the Neighbors.

Input : DAG $\mathcal{G} = (V, E, P)$

Output: DAG $\overline{\mathcal{G}} = (V, E, \overline{P})$

$AVG = (\sum_{v \in V} v)/|K|$

$\Delta = \sum_{i=1}^{K} (N(V_i) - AVG)^2$

while *True* **do**

  $found \leftarrow 0$

  for v *in* V **do**

    $pIndex \leftarrow v.p$

    $preds = Predecessors(v)$

    $succs = Successors(v)$

    if $offset = Gain(v, preds)$ **then**

      $Move(v, pIndex - offset)$

    end

    else if $offset = Gain(v, succs)$ **then**

      $Move(v, pIndex + offset)$

    end

  end

  if *found = 0* **then**

    break

  end

end

return $\mathcal{G} = (V, E, P)$

Algorithm 7: Helper *Gain* Function for Algorithm 6

```
def Gain(node,options):
    GainValues ← []
    for p in options do
        val = 2v² + 2v × avg × (N(V_{i+p}) − N(V_i))
        if val < 0 then
            GainValues.insert((p.index,val))
        end
    end
    return Max(GainValues).index
```

of a vertex, moving from a partition to a neighbour partition, while maintaining the acyclicity of the mapping, and reduces/keeps the value of Δ. To this end, we define the gain of a move as below:

$$\begin{aligned} Gain(v) =& 2v^2 + 2v \times avg \times (N(V_{i+1}) - N(V_i)) \\ & V_i \xrightarrow{v} V_{i+1} \, and \, i \leq K-1 \end{aligned} \tag{6.5}$$

$$\begin{aligned} Gain(v) =& 2v^2 + 2v \times avg \times (N(V_{i-1}) - N(V_i)) \\ & V_{i-1} \xleftarrow{v} V_i \, and \, i \geq 1 \end{aligned}$$

In Equation 6.5, $Gain(v)$ represents the change in value of Δ by moving the operation v, from its current partition to the previous/next partition. We obtained the $Gain(v)$ value in closed-form, from the formula of Δ.

A vertex is eligible for a move if it produces a negative $Gain(v)$ (i.e., reduces the value of Δ) by moving to the previous or the following partition. The algorithm repetitively finds and selects an eligible vertex from the graph (by calling $Predecessors(v)$ and $Successors(v)$, which will return eligible nodes from the nearby partitions), and

performs the move. In the case of multiple available vertices, we randomly choose one. The algorithm finds a local minimum based on the value of Δ. To better assist the algorithm, we can perform multiple (ℓ) topological sorting on the graph and then apply the simple move heuristic on all of them. Algorithm 6 presents a general form of this heuristic, where the set of *preds* and *succs* sets are the exact *previous* and *next* partitions (Each only includes one partition).

Extended Moves Heuristic: This heuristic extends the search for moving vertices between partitions beyond their neighbor partitions ($Predecessors(v)$ and $Successors(v)$ will return nodes beyond the exact neighbor partitions). The algorithm checks the farthest neighbor of a vertex on both directions (we call them V_A for the preceding node and V_B for the succeeding node) and exercises the possibility of moving the vertex to any of the partitions in-between while preserving the acyclicity constraints. Moving a vertex beyond its farthest neighbors creates a cycle. We define the gain of a move as below:

$$\begin{aligned} Gain(v) =& 2v^2 + 2v \times avg \times (N(V_{i+k}) - N(V_i)) \\ & V_i \xrightarrow{v} V_{i+k} \, and \, i + k \leq K \end{aligned} \tag{6.6}$$

$$\begin{aligned} Gain(v) =& 2v^2 + 2v \times avg \times (N(V_{i-k}) - N(V_i)) \\ & V_{i-k} \xleftarrow{v} V_i \, and \, i - k \geq 1 \end{aligned}$$

The gain in Equation 6.6 is following the same logic as the simple move heuristic, except it considers farther vertices for a possible move. Algorithm 6 presents this heuristic. The sets *preds* and *succs* present the set of predecessor and successor neighbors for the vertex to move. Further, the algorithm calculates the gain of moving a vertex to one of the predecessor or successor sets and moves to the partition that

gives the maximum potential gain.

6.3.2 Memory Scheduling

The execution order of the operations in a single partition can affect the total memory footprint of the design on the FPGA. Choosing a default sequence that only preserves DAG constraints can cause execution failure on devices with limited available memory. Also, it can increase overall energy consumption. As a result, a memory-aware scheduler can enable the RWNN execution on a broader range of FPGA(s).

Similar to the partitioning, the optimal execution ordering of the local DAG on an FPGA is an NP-Complete problem Bruno and Sethi (1976); Bernstein *et al.* (1989). The brute-force approach for computing all possible combinations has a computational complexity of $O(|V|!)$, which is exponential and expensive. Related works Bathie *et al.* (2020); Eyraud-Dubois *et al.* (2015); Agullo *et al.* (2016); Marchal *et al.* (2019) have studied memory-constrained DAG scheduling problems. All these works only aim to reduce the memory usage, below a specific threshold potentially. On the other hand, our design aims to find the best schedule possible (the least memory consumption) to enable the optimum power consumption and enable the RWNN to run on a broader range of FPGAs. To this end, we propose a *dynamic-programming (DP)* heuristic that achieves the optimal solution with the minimum computational complexity.

First, we define the structure of an optimal ordering as $\overline{\mathcal{O}}$. Also, we define $\overline{\mathcal{O}}_n$ as the optimal ordering for n number of nodes in the graph. The optimality of an ordering is realized through the minimum memory consumption during the execution. We also define $\mathcal{O}_n$ as an arbitrary execution order of n vertices. The solution amongst all $\mathcal{O}_n$, is $\overline{\mathcal{O}}_n$. As a principal of a DP algorithm, we need to define the relationship

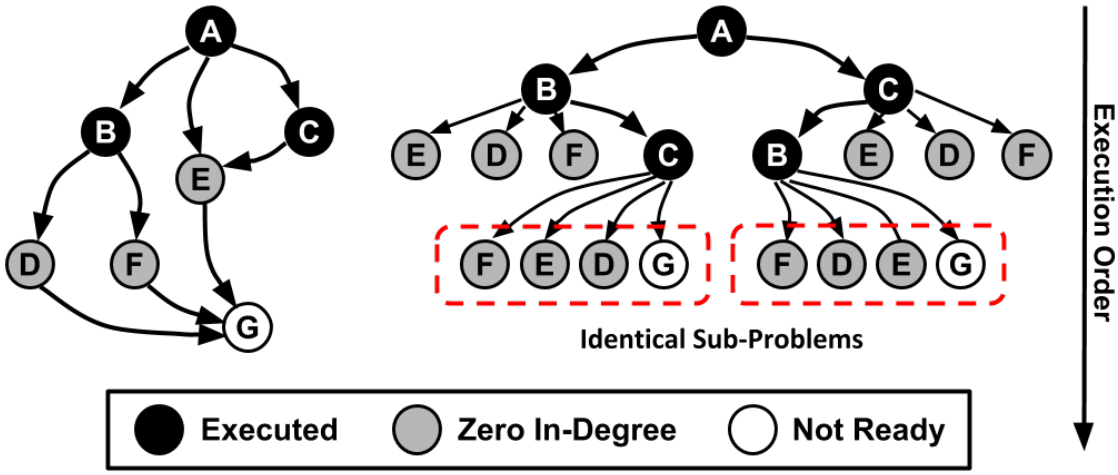


Figure 6.4: The Representation of How dynamic Programming Algorithm Covers Different Possibilities of Execution Orders, and Ultimately Figures Out the Optimal Order. Unlike the Brute-force Alternative, It Can Utilize the Memorization of the Duplicate Sub-problems, and Avoid Excessive Computation.

between $\overline{\mathcal{O}}_{n+1}$ and $\overline{\mathcal{O}}_n$. To build $\overline{\mathcal{O}}_{n+1}$, the algorithm should select the next operation to schedule, the operations where all their predecessors are already picked for the execution. The algorithm calculates $\mathcal{O}$ for all the possible moves and considers the operation that leads to the least memory consumption.

Considering the semantics of the above DP algorithm, and through our experiments, we observed long execution times while building the optimal ordering solutions. This is partly due to the redundant computations while building the $\mathcal{O}_n$ solutions. Figure 6.4 depicts an example where the algorithm faces an identical sub-problem while traversing the search space. Using the standard *Memoization* technique, the DP algorithm can improve the execution time.

As we mentioned above, the optimal ordering solution ($\overline{\mathcal{O}}$) is constructed based on the maximum memory footprint of the execution order. We extend the above generic DP algorithm to include the memory footprint constraint and define $\overline{\mathcal{O}}_n$, respectively.

In effect, we consider Mem_i and Mem_i^{top} as the memory footprint at step i, and maximum memory footprint up to step i. Our algorithm leverage memoization of the execution order EO_i (which is a order set of i nodes), and the respective Mem_i^{top}.

Algorithm 8 details our DP algorithm. We store the current and maximum memory footprint at each step, as Mem_i and Mem_i^{top}. We also memorize the execution order for the i vertices as $\mathcal{O}_i$. Also, we need to calculate the available *zero indegree* nodes and store it in $\mathcal{Z}^i$ to avoid repetitive calculations. We use $\mathcal{Z}^i$ as an index in our history, which enables the DP memoization. At each step (i), the algorithm fetches all possible sets of zero-indegree nodes ($\mathcal{Z}^i$, through different execution orders, up to step i). It iterates over all nodes in each $\mathcal{Z}^i$, and creates the new execution order $\mathcal{O}_{i+1}$ and the zero-indegree node-set $\mathcal{Z}^{i+1}$. It also calculates the memory footprint ($UpdateMemFP$) after adding the new node and removing the nodes with no outgoing vertices (which means their data can be evicted from the memory since no operation uses that data anymore). The memory footprint of each operation is calculated ($TotalWeight$) as the total number of parameters of that operation, considering the bit-precision of the model. Finally, we store (or update) the new zero-indegree in the history, which will be used in the next steps. After n steps, where n is the total number of vertices, the optimal execution order ($\overline{\mathcal{O}}$) can be derived by considering all available $\overline{\mathcal{O}}_n$s in the history, and choosing the order with the minimum Mem_n^{top}.

6.3.3 Scheduling-Aware Inter-FPGA Communication

The operations in RWNN can consume/feed data from/to any other operation with an arbitrary distance in the graph. This specific feature can affect the design of the accelerator on the FPGA. In all previous pipelined FPGA accelerators, the intermediate result by each FPGA is fed into the next FPGA. In contrast, in RWNNS, the neighbor (s) of a vertex may be assigned onto FPGA(s), multiple hops away from

Algorithm 8: Dynamic Programming Algorithm to Find The Optimal Execution Order.

Input : graph $\mathcal{G} = (V, E)$

Output: Optimal execution order $\overline{\mathcal{O}}$

`// Initization, ` $\mathcal{H}$ `stands for History`

$\mathcal{O}_0 = [\,]$, $Mem_0 \leftarrow 0$, $Mem_0^{top} \leftarrow 0$

$\mathcal{Z}^0 \leftarrow ZeroIndegree(\mathcal{O}_0, \mathcal{G})$

$\mathcal{H}[\mathcal{Z}^0] \leftarrow (\mathcal{O}_0, Mem_0, Mem_0^{top})$

`// Performing the DP steps`

for i *in* $(0, n)$ **do**

  `// possible orders for i nodes`

  for $(\mathcal{Z}^i, (\mathcal{O}_i, Mem_i, Mem_i^{top}))$ *in* $\mathcal{H}$ **do**

    for *node in* $\mathcal{Z}^i$ **do**

      $\mathcal{O}_{i+1} \leftarrow \mathcal{O}_{i+1} + node$

      $\mathcal{Z}^{i+1} \leftarrow ZeroIndegree(\mathcal{O}_{i+1}, \mathcal{G})$

      `// Current memory footprint`

      $Mem_{i+1} \leftarrow TotalWeight(node) + Mem_i$

      $Mem_{temp}^{top} \leftarrow Mem_{i+1}$

      $Mem_{i+1}^{top} \leftarrow UpdateMemFP(node, \mathcal{O}_{i+1})$

      `// Footprint memorization`

      $StoreOrder(Mem_{i+1}^{top}, Mem_{i+1}, \mathcal{Z}^{i+1}, \mathcal{O}_{i+1}, \mathcal{H})$

    end

  end

end

return $\overline{\mathcal{O}} \leftarrow \mathcal{O}_n$

Algorithm 9: Helper *TotalWeight* Function for Algorithm 8

```
// total number of params in a node
def TotalWeight(node):
    return ∏_{i=1}^{node.dims} (node[i].length)
```

Algorithm 10: Helper *UpdateMemFP* Function for Algorithm 8

```
// Updating the memory footprint
def UpdateMemFP(node,O,Mem_{i+1}):
    for p in node.parents do
        if p is in zeroOutdegree(O,G) then
            Mem_{i+1} ← Mem_{i+1} - TotalWeight(node)
        end
    end
```

Algorithm 11: Helper *StoreOrder* Function for Algorithm 8

```
// Store the order in the history
def StoreOrder(Mem,Mem^top,Z,O,H):
    if Mem_{i+1}^top ≤ H[Z^{i+1}][Mem_{i+1}^top] then
        H[Z^{i+1}] ← (O_{i+1},Mem_{i+1},Mem_{i+1}^top)
    end
```

the source FPGA. As a result, the current design cannot deliver the intermediate data to the designated FPGA for RWNNs.

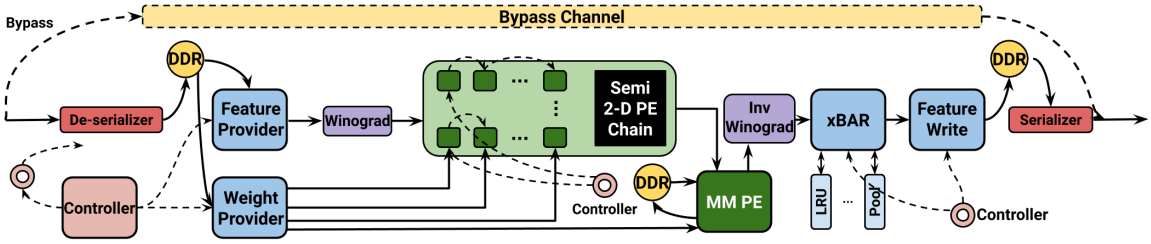


Figure 6.5: CNN Accelerator Architecture.

To address the above issue, the accelerator design should be extended to support arbitrary data delivery. In other words, each FPGA should decide whether to consume the incoming data or bypass it to a succeeding FPGA. Support for this feature requires two important extensions to the previous multi-FPGA accelerator design: (1) each FPGA should support a direct pipe from the input to the output physical channel, and (2) FPGAs should contain an extra logic that determines the correct destination of an incoming set of data. The first extension can be added to the design, using a *De-Multiplexer*. The incoming data first hits the de-multiplexer and then follows the designated path. The logic utilizes some predetermined information for the second extension, which is decided during the scheduling phase. Based upon the topology of the graph, partition of each vertex, and the execution order of the vertices in each partition, the scheduler determines the destination of each input data on each FPGA. This logic enables the correct path of the de-multiplexer, which directs the input straight to the output, or the computational pipeline. This information is being sent to each FPGA while bootstrapping the FPGAs with the operation parameters. Further, the FPGAs follow this instruction for proper consumption/bypass of the incoming data. Figure 6.5 represents the extended FPGA accelerator design, with support for RWNNs. The bypass channel, combined with the de-multiplexer logic,

is added to the design from related work Biookaghazadeh *et al.* (2020) to support RWNNs.

The extensive usage of depth-wise separable convolutions in RWNNs requires support from the accelerator. Unfortunately, previous FPGA accelerator designs only focus on traditional convolutions, especially the systolic array designs, which are proven to be the most efficient. One might simulate depth-wise convolutions with traditional convolutions, but it contradicts the primary purpose of reducing the total MAC operations. We extended the previous FPGA design Biookaghazadeh *et al.* (2020) to support depth-wise convolutions. Our extended design requires to make minimum modification to the original design to avoid resource over-consumption. Also, it needs to guarantee maximum performance for both conventional and depth-wise convolutions.

The extended design specifically made two modifications in the data provider and the processing elements (PE) systolic array to enable depth-wise convolution. Here we discuss the specific changes in more depth:

Data Provider: The current design reads a block of size $Width \times Height \times IN\_CH\_SIZE$, where $Width$ is a constant value (in our case, it is eight), to increase data reuse along the spatial dimension. The data provider reads all blocks along the channel dimension first and then proceeds through the width and height dimensions. Unfortunately, it will lead to frequent weight loading on each PE (since each filter is applied on a single input channel, but not all). To avoid this issue, the provider needs to scan the data through the spatial dimension (width/height) rather than the temporal dimension used in the previous design.

Weight Provider and PEs: In the original design, the weight provider loads each PE (32 PEs in total) with features of a single output channel. Each weight feature is of size $Input_ch \times width \times height$. The whole feature is convolved with the input

to generate the output data for one channel. In comparison, depth-wise convolutions come with $Input_c h$ size of one for each weight feature, and they only get convolved with one input channel of data. One can simply reduce the IN_CH_SIZE from the data provided to one to enable 2-D convolutions seamlessly but at the cost of less parallelization and lower performance. To avoid this issue, we load multiple output features into each PE and perform the same multiplication-accumulation on the input plate of data. The only difference is the accumulation phase, where instead of adding all multiplied data from all weight channels, we separately accumulate the data for each channel.

6.3.4 Operation Division

Operations in an RWNN (or a DNN) graph can embody different computational intensities. In some cases, the large gap of intensity can stop the scheduler from finding a good balance of load between the partitions, regardless of the partitioning scheme. To alleviate this problem, we propose *Operation Division* technique. This technique is rooted in graph transformation techniques, which take advantage of the distributive, associative, and commutative properties of the neural network operations, as explained in chapter 6.2. Doing so breaks down large operations into a set of small parallel operations and ultimately reduces the standard deviation of the overall computational intensity between all operations. Figure 6.6 represents a sample of breaking down large operations into multiple small operations, following a *concatenation*.

Efficient operation division is necessary to obtain the optimal partitioning scheme. Toward this goal, we proposed Algorithm 12. Our algorithm's main idea is to converge the node weights into a global average weight through node division. We first calculate the global average weight by dividing the total graph weight by the number of nodes.

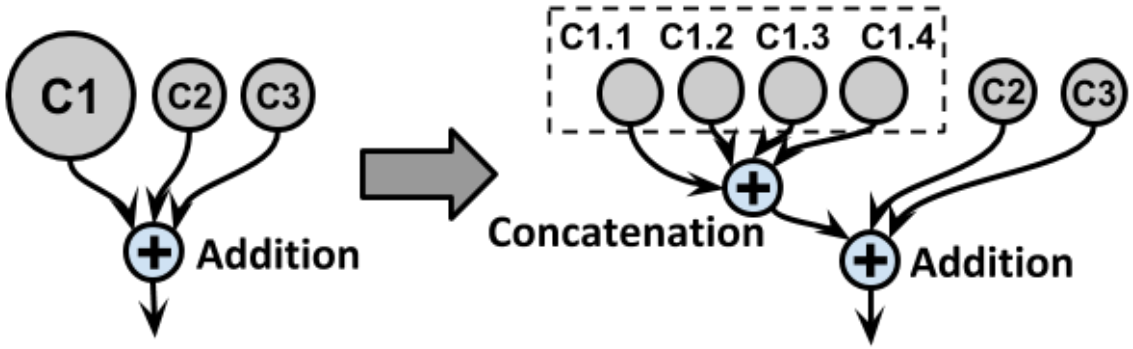


Figure 6.6: Splitting a Large Operation Into Multiple Smaller Operations, Following a Concatenation of the Results.

Second, we iterate over the nodes and decide whether divisions reduce the distance of the new node weights and the global average weight. Further, we replace the node with the new nodes and rewire the disconnected edges. The new nodes' output needs to be added by a new addition operation before being forwarded to the successor node.

6.4 Evaluation

To confirm and quantify our novel scheduler's benefits, we designed and conducted experiments on our multi-FPGA platform.

All experiments were conducted on a network of Intel Fog Reference Design units each equipped two Nallatech 385A FPGA acceleration cards (Intel A10 GX1150 FPGA), an Intel Xeon E5-1275 v5 CPU, and 32GB of main memory. The OpenCL kernels for FPGAs were compiled using Intel FPGA SDK for OpenCL (version 19.1) with Nallatech p385a_sch_ax115 board support packages (BSP).

We selected four RWNN architectures for evaluation throughout the rest of the paper. Table 6.1 summarizes the specifications of these networks. These networks include 11 or 22 nodes, and two classical graph generators, Erdos-Renyi (ER) Erdős

Algorithm 12: Operation Division Algorithm.

Input : graph $\mathcal{G} = (V, E)$

Output: graph $\overline{\mathcal{G}} = (\overline{V}, \overline{E})$

`// Initialization`

$W_{total} = 0$, $W_{avg} = 0$

for *node in* V **do**
  $W_{total} \leftarrow W_{total} + node.fmas$

end

$W_{avg} \leftarrow W_{total}$ / $len(V)$

`// Divide operations, if necessary`

for *node in* V **do**
  $NumDivs = CalcNumDivs(node.fmas, node.ich, W_{avg})$
  if $NumDivs \neq 1$ **then**
    $Divide(\mathcal{G}, node, numDivs)$
  end

end

return $\overline{\mathcal{G}} \leftarrow \mathcal{G}$

and Rényi (1960) and Watts-Strogatz (WS) Watts and Strogatz (1998). The number of nodes does not represent the total number of operations but rather a graph generator parameter.

Finally, all RWNN architectures are performing inference on images of size *3C* $\times$*224W* $\times$*224H*.

6.4.1 FPGA Design Extension

As we discussed in Chapter 6.3.3, the available FPGA implementation lacks support for RWNNs. More precisely, the new FPGA design should be knowledgeable

Algorithm 13: Helper *CalcNumDivs* Algorithm for the Operation Division Algorithm 12.

```
// Return optimal number of divisions
def CalcNumDivs(fmas, ICH, W_avg):
    OptNumDivs = 1
    offset = abs(fmas - W_avg)
    while true do
        newICH = ceil(ICH / (OptNumDivs + 1))
        W_new = (newICH / ICH) × fmas
        if offset ≤ abs(fmas - W_new) then
            offset = abs(fmas - W_new)
            OptNumDivs ← OptNumDivs + 1
        else
            break
        end
    end
    return OptNumDivs
```

about (1) proper execution order of the operations, (2) allocation/eviction of the memory regions for the input/output of each operation, and (3) consumption/redirection of the incoming data from the input I/O channel.

This section presents the extra logic overhead to support the three new features of the accelerator design. Table 6.2 shows the resource consumption of the new design compared to the baseline. We can observe a slight increase in BRAM, LUT, and FF usage, around 7%, 15%, and 17%. Also, we can observe an almost the same clock frequency, which can help preserve a similar performance. As a result, the extended design can easily fit the same FPGA while providing similar performance.

Algorithm 14: Helper *Divide* Algorithm for the Operation Division Algorithm 12.

```
def Divide(G, node, numDivs):
    node.removeConnections(node.incomings())
    node.removeConnections(node.outgoings())
    G.remove(node)
    for i in range(numDivs) do
        start = i × (node.ich/numDivs)
        end = (i + 1) × (node.ich/numDivs)
        newNode = node[start : end][:][:]
        newNode.incomings() = node.incomings()
        newNode.outgoings() = node.outgoings()
        G.add(newNode)
    end
```

6.4.2 Partitioning

This chapter evaluates our partitioning algorithm's effectiveness in balancing the load among the FPGAs in the pipeline. As mentioned before, an improper balance can affect the overall throughput. To measure the effectiveness, we first need to choose a metric to evaluate the partitioning. We consider the partition's deviation with the most prominent weight from the optimal average weight (which can provide the optimal throughput) calculated as $W_{total}/num\_partitions$. The deviation is calculated as Equation 6.7, where W_{avg} is an optimal average weight for each FPGA and W_{max} is the FPGA with maximum assigned weight after the partitioning. Since the throughput is bounded by the FPGA with the largest weight, the smaller deviation means less overhead on the pipeline.

Name	Generator	Number of Nodes
RWNN-1	ER	11
RWNN-2	ER	22
RWNN-3	WS	11
RWNN-4	WS	22
ER-Specific Configuration Parameters		
Edge Creation Prob. (p)		0.4
WS-Specific Configuration Parameters		
Edge Creation Prob. (p)		0.4
Number of Nearest Neighbors (k)		4

Table 6.1: RWNN architecture configurations.

Design	Clock Freq (MHz)	DSP Util. (Used / Total)	BRAM Util. (Used / Total)	LUT Util (Used / Total)	FF Util. (Used / Total)
Baseline	212	1.4K / 1.5K	1.3K / 2.7K	290K / 854K	762K / 1.7M
Xtended	210	1.4K / 1.5K	1.4K / 2.7K	335K / 854K	894K / 1.7M

Table 6.2: Resource utilization of the baseline FPGA design vs. the extended FPGA design.

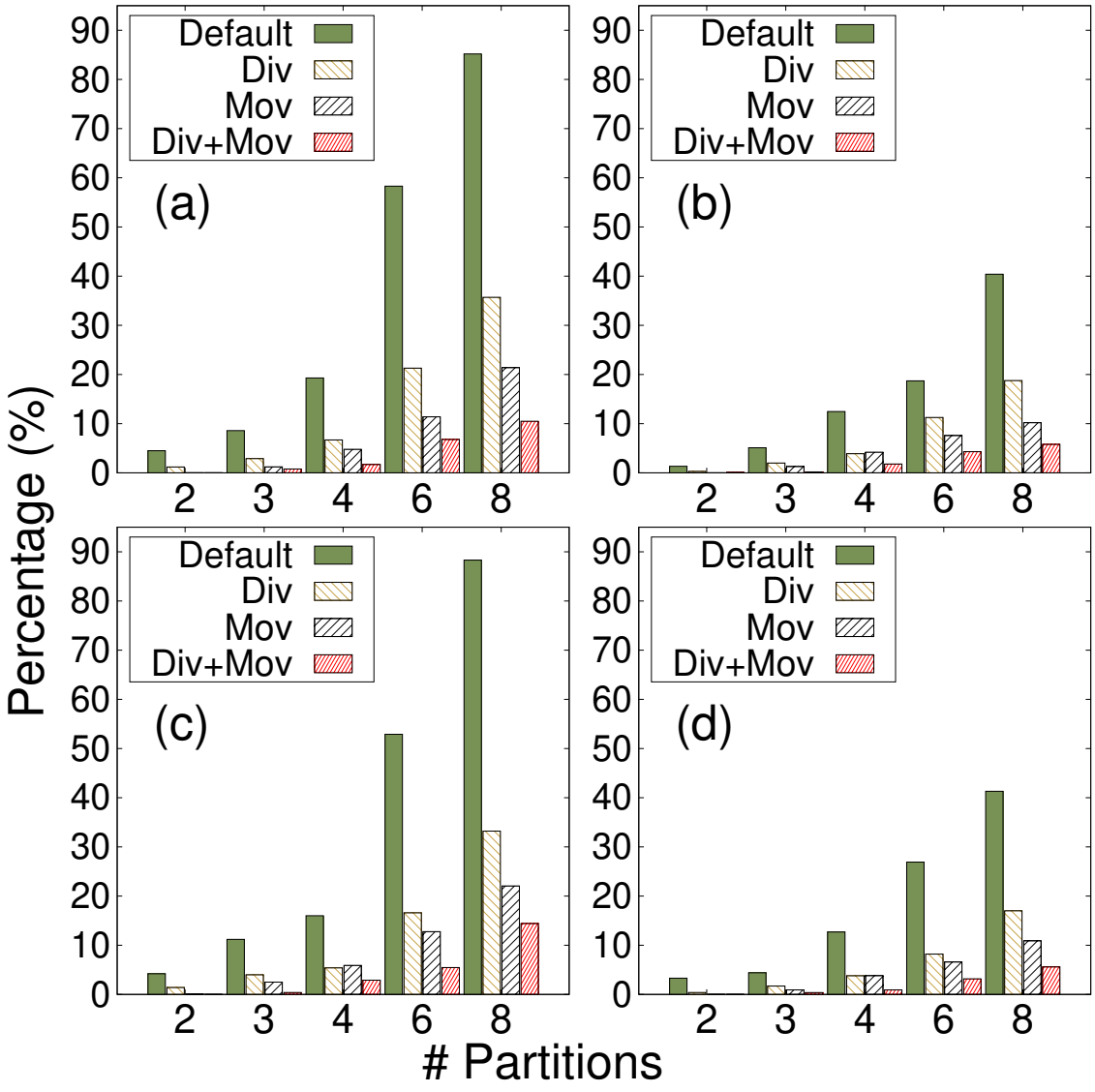


Figure 6.7: Deviation from the perfect average weight for: (a) RWNN-1, (b) RWNN-2, (c) RWNN-3, and (d) RWNN-4, for 4 different partitioning algorithms. **Div** and **Mov** stand for Division and Moving.

$$Deviation(\%) = (abs(W_{avg} - W_{max}))/W_{avg} \times 100 \tag{6.7}$$

We first compared the effectiveness of our move heuristics, simple move heuristic and extended move heuristic. Figure 6.8 depicts the comparison between these two

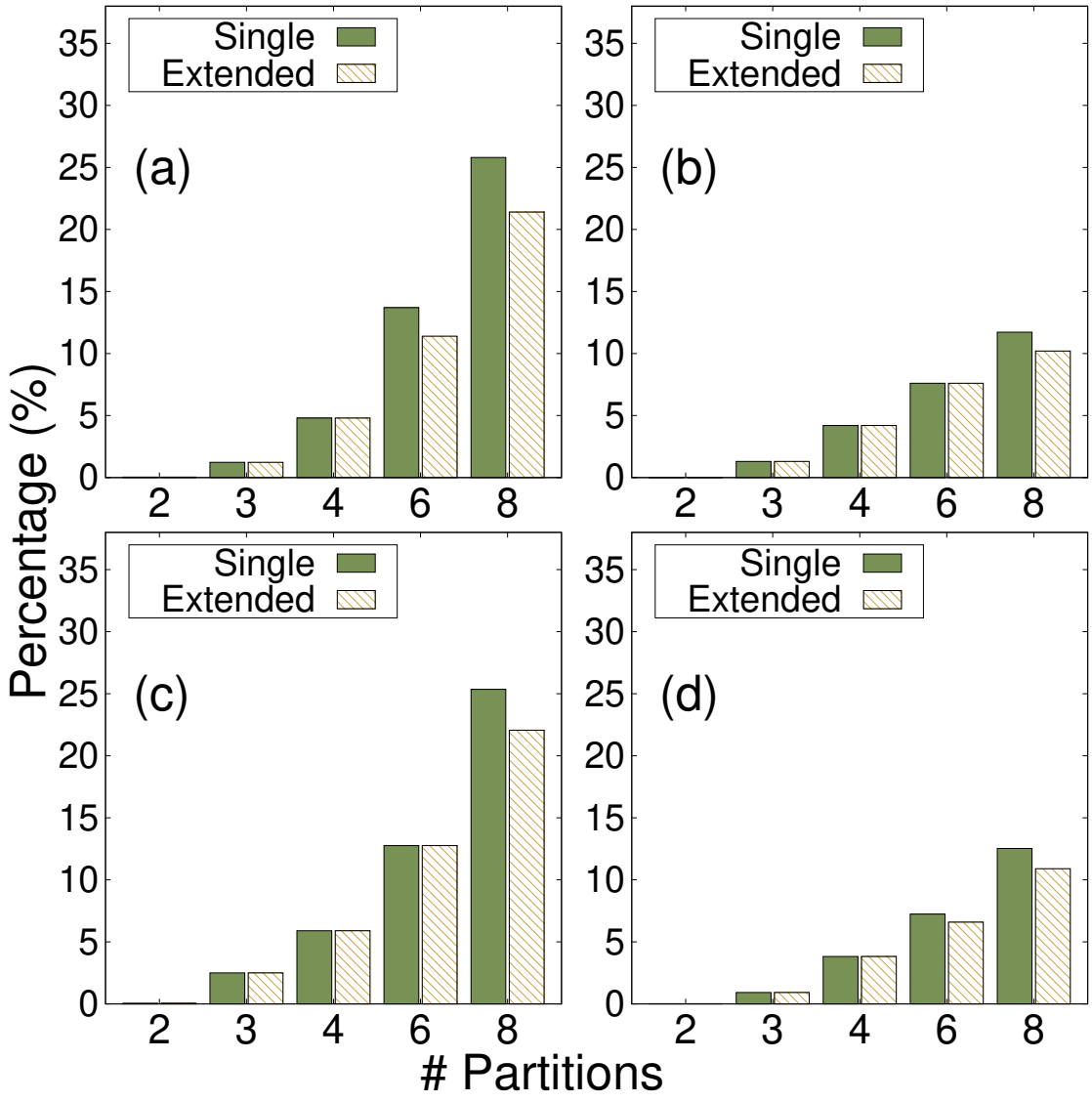


Figure 6.8: Deviation from the perfect average weight for: (a) RWNN-1, (b) RWNN-2, (c) RWNN-3, and (d) RWNN-4, for single-move and extended-move heuristics.

heuristics for different number of FPGAs. Based on the results, the extended-move heuristics is outperforming the single-move up to 20%, while balancing the load along the FPGAs. As a result, we stick to extended-move heuristic, throughout the rest of this paper.

Figure 6.7 compares the deviation in four different partitioning algorithms: (1) default, (2) only division of operators (Div), (3) moving operators between partitions (mov), and (4) the combination of division and moving. Need to mention, we used extended moves heuristic. The X-Axis is the number of FPGAs, and the Y-Axis is the deviation percentage from the optimal average weight. Our experiments include four different graph configurations. The results are showing that our partitioning algorithm can reduce the deviation between 2.4x to 8.1x. Such reduction helps the deployment to achieve a much higher throughput on the pipeline.

Another observation is the effectiveness of different partitioning techniques. Based on our observations, while moving operations between different partitions and dividing operations both reduce the deviation (up to x3.9 for moving and up to 2.3x for division), the former technique is much more effective. This is because the accelerator supports a specific granularity of operation configuration, making any operation with a smaller configuration have the same overhead. Finally, we realized that combining both techniques can provide the maximum possible benefit by reducing the deviation up to 8.1x.

Figure 6.9 presents the throughput of the FPGA pipeline (with different numbers of FPGAs) under different partitioning strategies. The X-Axis is the number of FPGAs, and the Y-Axis is the throughput of the pipeline (Img/second). The figure also presents the throughput that can be achieved in a perfect condition. Our partitioning strategy to divide the operations and move them between the partitions can obtain around 81.1% better throughput than the default strategy.

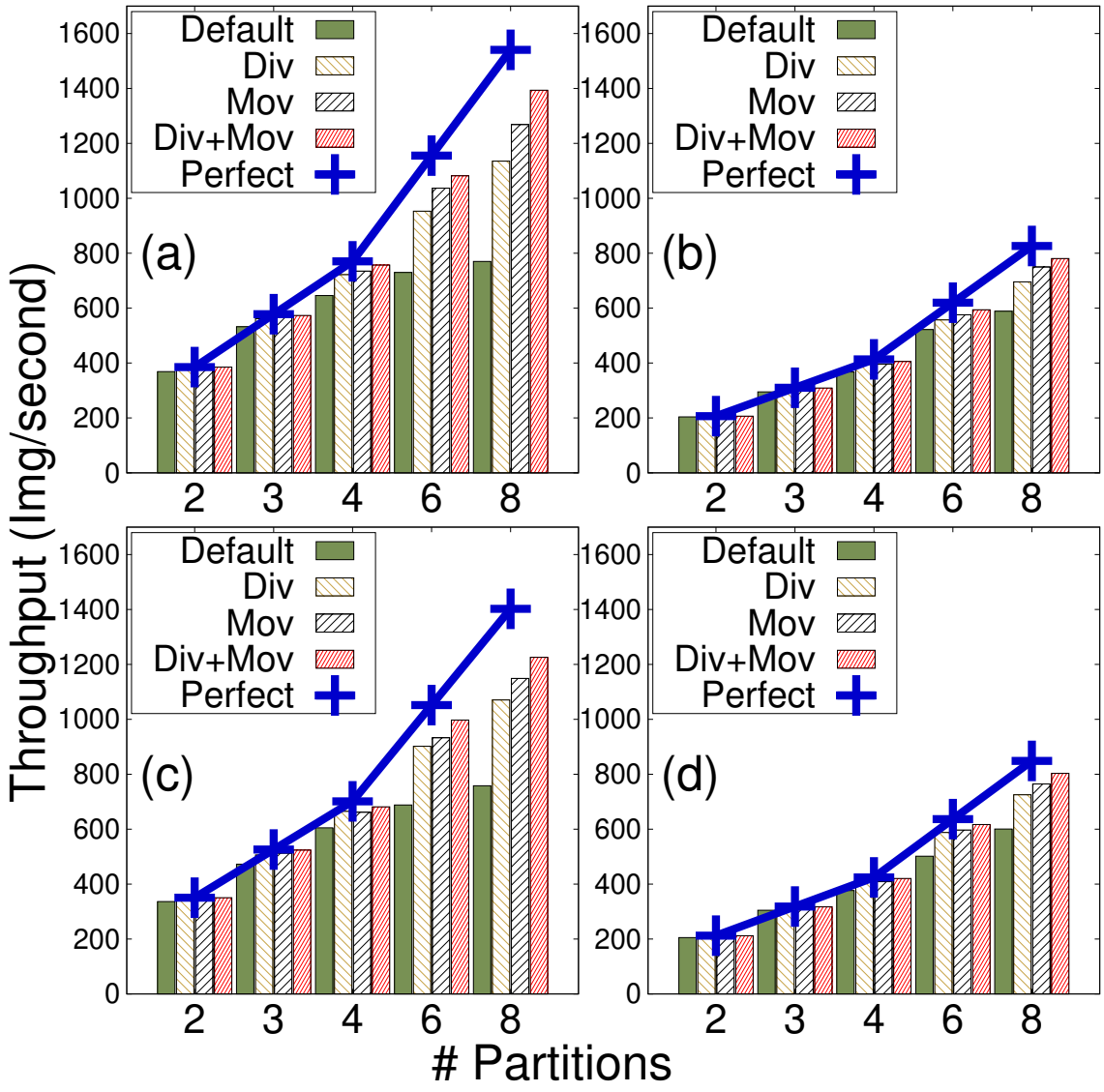


Figure 6.9: Throughput of the FPGA pipeline with different partitioning strategies, including the perfect partitioning for configurations: (a) RWNN-1, (b) RWNN-2, (c) RWNN-3, and (d) RWNN-4.

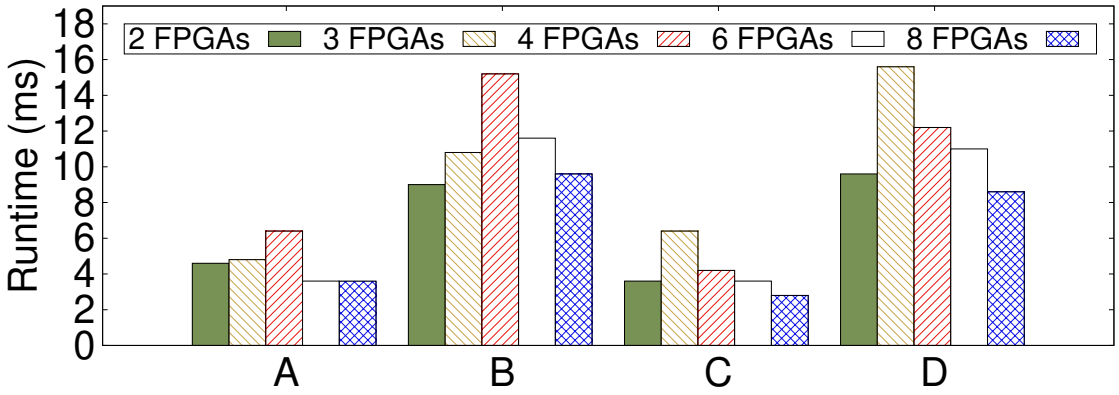


Figure 6.10: Execution time overhead of our partitioning algorithm, for different architectures: (A) RWNN-1, (B) RWNN-2, (C) RWNN-3, and (D) RWNN-4.

There are two main reasons that prevent any strategy from obtaining linear speedup with increasingly number of FPGAs. First, the finest granular load of work is processing one channel of data, which stops the partitioning strategy to achieve perfect load balance. Second, and most importantly, while dividing the input channels of convolutions, we may need to add some extra empty channels (like padding in the normal convolution) to match them with the hard-coded number of input channels in the FPGA design (IN_CH_SIZE). The above limitations prevent any strategy from achieving the perfect load balance or the ideal throughput.

Finally, Figure 6.10 shows the overhead of our partitioning algorithm for all RWNNs and different numbers of FPGAs. In all our cases, the overhead is below 20ms. This is negligible compared to the overall network execution overhead. Also, partitioning is a one-time process and will not affect the pipeline performance.

6.4.3 Memory Scheduling

In this chapter, we evaluate the potential of our memory scheduler in optimizing the maximum memory consumption. As mentioned in Chapter 6.3.2, different exe-

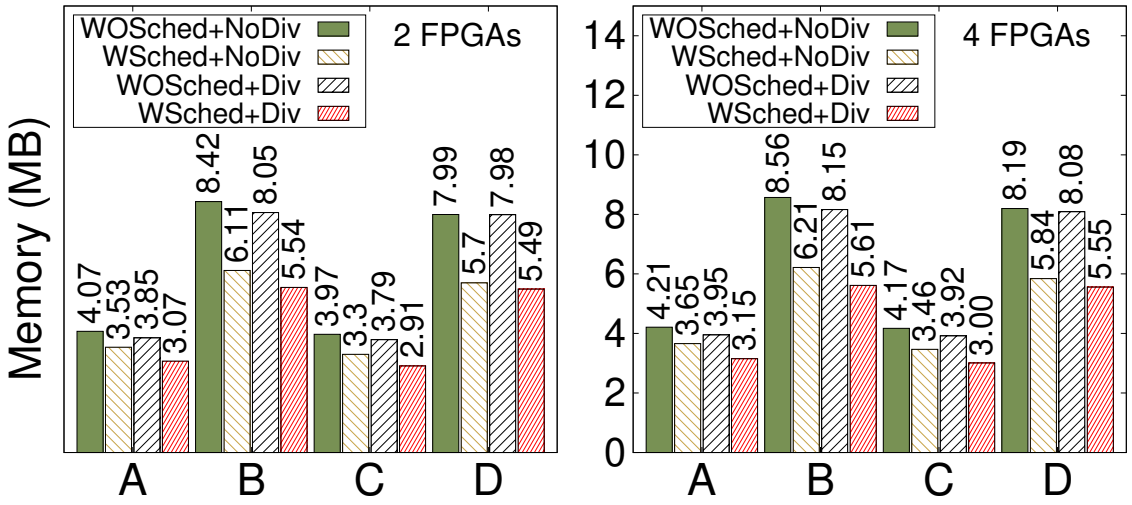


Figure 6.11: Memory consumption of the RWNN on two and four FPGAs. There four different bars, representing the combination of with or without scheduling (WSched and WOSched) and with or without operation division (Div and NoDiv).

cution orders of the operations on an FPGA can have different maximum memory footprints. On FPGAs with limited off-chip memory, some execution orders may fail due to out-of-memory issues. As a result, an optimal memory-aware scheduler is necessary to enable RWNN execution on a broader range of devices.

Figure 6.11 depicts the maximum memory utilization, with and without our memory-aware scheduler. Experiments were performed on two and four FPGA configurations. On X-Axis, we have four different RWNNs, as described in Table 6.1. Y-Axis represents the memory consumption for each RWNN and a specific scheduling setting. Three observations are available from Figure 6.11. First, our memory scheduler can reduce the memory footprint by up to 34%. In our case, the scheduler has better opportunities with bigger neural networks (B and D, against A and C). Second, our operation division approach can slightly improve the memory footprint (up to 5%), even without the scheduler. With the operation division, the scheduler can break up part of the large operations from the FPGA, which can lead to a lower

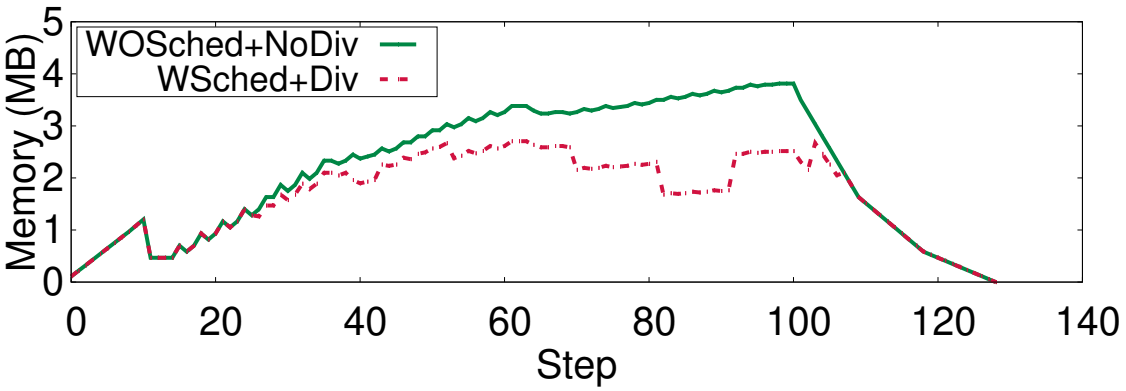


Figure 6.12: Memory footprint during the execution at each step (execution of an operation), with two configurations: (1) without operation division and scheduling, (2) with operation division and scheduling.

memory footprint. Third, increasing the number of FPGAs reduces the effectiveness of the memory scheduler. It stems from having fewer scheduling options on each FPGA (with a smaller number of operations per FPGA) and less opportunity for memory reduction. For example, for a single experiment, the memory footprint on a four-FPGA configuration is roughly 2% higher than a two FPGA configuration.

Figure 6.12 presents a detailed memory consumption footprint while running RWNN-1 with operation division on a single FPGA, with and without memory scheduling. Each step in the X-Axis is an operation execution (including new memory (de)allocations). We can observe that our memory scheduler reduces the memory consumption throughput the execution.

6.5 Conclusions and Future Works

In this chapter we designed *Piper* to efficiently schedule RWNNs on multi-FPGA platforms. *Piper* comes with three main contributions. First, it determines the optimal partitioning of an RWNN on a multi-FPGA platform. Second, it reduces

the memory footprint on each FPGA, by discovering the optimal execution plan. Third, it extends the available neural network accelerator design on FPGAs to support arbitrary data consumption/routing, which is a unique feature of RWNN graphs. Finally, we evaluated our design and shown the effectiveness of *Piper*.

Chapter 7

CONCLUSIONS

In this thesis we studied the suitability of FPGAs for accelerating edge workloads, specially the emerging neural networks. Here we summarize our findings in each chapter:

- First, we studied the general suitability of FPGAs for IoT workloads on edge servers. Our results confirm the superiority of FPGAs over GPUs with respect to: (1) providing workload- insensitive throughput; (2) adaptiveness to both spatial and temporal parallelism at fine granularity; and (3) better energy efficiency and thermal stability. Based on our observations, we argue that FPGAs should be considered a replacement or complementary solution for current processors on edge servers.

- Second, we studied the feasibility of different accelerators for handling different loop patterns, through design of Loopy. We identified and analyzed five common loop patterns, along with the key configuration parameters in these patterns. We then studied the acceleration opportunities for these loop patterns and how the loop configurations and accelerator platforms affect the effectiveness of acceleration. Using Loopy, developers can gain a good understanding of the acceleration potential of their algorithms on different platforms, without having to implement them for any specific platform, based on the loop patterns that these algorithms embody.

  Loopy provides an important first step towards the optimized use of accelerators for diverse applications in heterogeneous computing systems. Based on

Loopy, we will be able to study the combination of the loop patterns and their accelerations ability in heterogeneous computing systems in our future works.

- Third, we demonstrated a high-throughput multi-FPGA acceleration solution for a wide variety of CNNs. We first extended the DLA Aydonat *et al.* (2017) architecture to achieve lower latency and better resource utilization, for single FPGA configuration. further, we extended the architecture to support 3D convolutions for the video understanding applications. It includes studying the optimal deployment of the 3D convolutions on the semi-1D systolic array. further, we enabled communication between the FPGAs to support multi-FPGA setups. Finally, we developed an algorithm for automatic deployment of the CNN layers onto the FPGAs in the multi-FPGA setup. Our results show that utilizing multiple FPGAs can linearly increase the overall throughput of the CNN inference. Also, optimizing the PE systolic array and FC operations can lead to better area utilization (up to 25%) and higher overall throughput (up to 24%), compared to the state-of-the-art CNN accelerator. Finally, we studied the efficient mapping of 3D convolutions on our novel semi-1D systolic array to achieve the highest overall throughput.

- Fourth and final, we designed *Piper* to efficiently schedule RWNNs on multi-FPGA platforms. *Piper* comes with three main contributions: (1) It determines the optimal partitioning of an RWNN on a multi-FPGA platform, (2) It reduces the memory footprint on each FPGA, by discovering the optimal execution plan, and (3) It extends the available neural network accelerator design on FPGAs to support arbitrary data consumption/routing, which is a unique feature of RWNN graphs. Finally, we evaluated our design and shown the effectiveness of *Piper*.

REFERENCES

The OpenCL Specification, Khronos OpenCL Working Group, rev. 19 (2012).

Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning", in "12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)", pp. 265–283 (2016).

Abbas, N., Y. Zhang, A. Taherkordi and T. Skeie, "Mobile edge computing: A survey", IEEE Internet of Things Journal **5**, 1, 450–465 (2017).

Abdelfattah, M. S., D. Han, A. Bitar, R. DiCecco, S. O'Connell, N. Shanker, J. Chu, I. Prins, J. Fender, A. C. Ling *et al.*, "Dla: Compiler and fpga overlay for neural network inference acceleration", in "2018 28th International Conference on Field Programmable Logic and Applications (FPL)", pp. 411–4117 (IEEE, 2018).

Adelantado, F., X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui and T. Watteyne, "Understanding the limits of lorawan", IEEE Communications magazine **55**, 9, 34–40 (2017).

Agullo, E., P. R. Amestoy, A. Buttari, A. Guermouche, J.-Y. l'Excellent and F.-H. Rouet, "Robust memory-aware mappings for parallel multifrontal factorizations", SIAM Journal on Scientific Computing **38**, 3, C256–C279 (2016).

Ananthanarayanan, G., P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath and S. Sinha, "Real-time video analytics: The killer app for edge computing", Computer **50**, 10, 58–67 (2017).

Ancourt, C. and F. Irigoin, "Scanning polyhedra with do loops", in "Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming", pp. 39–50 (1991).

Arsham, H. and A. Kahn, "A simplex-type algorithm for general transportation problems: An alternative to stepping-stone", Journal of the Operational Research Society **40**, 6, 581–590 (1989).

Arulkumaran, K., M. P. Deisenroth, M. Brundage and A. A. Bharath, "Deep reinforcement learning: A brief survey", IEEE Signal Processing Magazine **34**, 6, 26–38 (2017).

Aydonat, U., S. O'Connell, D. Capalija, A. C. Ling and G. R. Chiu, "An opencl™ deep learning accelerator on arria 10", in "Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays", pp. 55–64 (ACM, 2017).

Baghdadi, R., J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code", in "2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)", pp. 193–205 (IEEE, 2019).

Banerjee, U., *Dependence analysis*, vol. 3 (Springer Science & Business Media, 1997).

Barkatullah, J. and T. Hanke, "Goldstrike 1: Cointerra's first-generation cryptocurrency mining processor for bitcoin", IEEE micro **35**, 2, 68–76 (2015).

Bastoul, C., "Code generation in the polyhedral model is easier than you think", in "Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.", pp. 7–16 (IEEE, 2004).

Bathie, G., L. Marchal, Y. Robert and S. Thibault, "Revisiting dynamic dag scheduling under memory constraints for shared-memory platforms", in "2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)", pp. 597–606 (IEEE, 2020).

Beckmann, K. and O. Dedi, "sdds: A portable data distribution service implementation for wsn and iot platforms", in "2015 12th International Workshop on Intelligent Solutions in Embedded Systems (WISES)", pp. 115–120 (IEEE, 2015).

Bellman, R., "Dynamic programming", Science **153**, 3731, 34–37 (1966).

Bellman, R., *Dynamic programming* (Courier Corporation, 2013).

Belviranli, M. E., P. Deng, L. N. Bhuyan, R. Gupta and Q. Zhu, "Peerwave: Exploiting wavefront parallelism on gpus with peer-sm synchronization", in "Proceedings of the 29th ACM on International Conference on Supercomputing", pp. 25–35 (ACM, 2015).

Bernstein, D., M. Rodeh and I. Gertner, "On the complexity of scheduling problems for parallel/pipelined machines", IEEE Transactions on computers **38**, 9, 1308–1313 (1989).

Biookaghazadeh, S., "LoopBench", `https://github.com/saman-aghazadeh/shoc-fpga/` (2020).

Biookaghazadeh, S., P. Kumar Ravi and M. Zhao, "Toward multi-fpga acceleration of the neural networks", ACM Journal on Emerging Technologies in Computing Systems (JETC) **16**, 2, 1–35 (2020).

Biookaghazadeh, S., M. Zhao and F. Ren, "Are fpgas suitable for edge computing?", in "{USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18)", (2018).

Boman, E. G., K. D. Devine and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2d graph partitioning", in "Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis", pp. 1–12 (2013).

Boutros, A., S. Yazdanshenas and V. Betz, "You cannot improve what you do not measure: Fpga vs. asic efficiency gaps for convolutional neural network inference", ACM Transactions on Reconfigurable Technology and Systems (TRETS) **11**, 3, 20 (2018).

Boyer, M., J. Meng and K. Kumaran, "Improving GPU performance prediction with data transfer modeling", in "Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International", pp. 1097–1106 (IEEE, 2013).

Bruno, J. and R. Sethi, "Code generation for a one-register machine", Journal of the ACM (JACM) **23**, 3, 502–510 (1976).

Bui, T. N., S. Chaudhuri, F. T. Leighton and M. Sipser, "Graph bisection algorithms with good average case behavior", Combinatorica **7**, 2, 171–191 (1987).

Buluc, A. and K. Madduri, "Graph partitioning for", Graph Partitioning and Graph Clustering **588**, 83 (2013).

Cai, H., L. Zhu and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware", arXiv preprint arXiv:1812.00332 (2018).

Callahan, D., J. Dongarra and D. Levine, "Vectorizing compilers: A test suite and results", in "Supercomputing'88.[Vol. 1]., Proceedings.", pp. 98–105 (IEEE, 1988).

Camilus, K. S. and V. Govindan, "A review on graph based segmentation.", International Journal of Image, Graphics & Signal Processing **4**, 5 (2012).

Chang, J.-W., K.-W. Kang and S.-J. Kang, "An energy-efficient fpga-based deconvolutional neural networks accelerator for single image super-resolution", IEEE Transactions on Circuits and Systems for Video Technology (2018).

Chapuis, G., S. Eidenbenz and N. Santhi, "Gpu performance prediction through parallel discrete event simulation and common sense", in "Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools", pp. 204–211 (ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016).

Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing", in "2009 IEEE International Symposium on Workload Characterization (IISWC)", pp. 44–54 (Ieee, 2009).

Chen, D. and D. Singh, "Fractal video compression in opencl: An evaluation of cpus, gpus, and fpgas as acceleration platforms", in "2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)", pp. 297–304 (IEEE, 2013).

Chen, T., T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning", in "13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)", pp. 578–594 (2018).

Cheng, H.-P., T. Zhang, Y. Yang, F. Yan, S. Li, H. Teague, H. Li and Y. Chen, "Swiftnet: Using graph propagation as meta-knowledge to search highly representative neural architectures", arXiv preprint arXiv:1906.08305 (2019).

Chiang, M. and T. Zhang, "Fog and iot: An overview of research opportunities", IEEE Internet of Things Journal **3**, 6, 854–864 (2016).

Chiou, D., "The microsoft catapult project", in "2017 IEEE International Symposium on Workload Characterization (IISWC)", pp. 124–124 (IEEE Computer Society, 2017).

Cisco, "Fog Computing and Internet of Things", Tech. rep., Cisco (2015).

Cisco, "Cisco UCS C220", `https://www.optiodata.com/` (2019).

Claypool, M. and K. Claypool, "On latency and player actions in online games", (2006).

Cong, J., Z. Fang, Y. Hao, P. Wei, C. H. Yu, C. Zhang and P. Zhou, "Best-effort FPGA programming: A few steps can go a long way", arXiv preprint arXiv:1807.01340 (2018a).

Cong, J., Z. Fang, M. Lo, H. Wang, J. Xu and S. Zhang, "Understanding performance differences of fpgas and gpus", in "2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)", pp. 93–96 (IEEE, 2018b).

Cong, J. J. and J. R. Shinnerl, *Multilevel optimization in VLSICAD*, vol. 14 (Springer Science & Business Media, 2013).

Cousot, P. and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program", in "Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages", pp. 84–96 (1978).

Da Silva, B., A. Braeken, E. H. D'Hollander and A. Touhafi, "Performance modeling for FPGAs: extending the roofline model with high-level synthesis tools", International Journal of Reconfigurable Computing **2013**, 7 (2013).

Danalis, A., G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite", in "Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units", pp. 63–74 (ACM, 2010).

Dean, J., "Machine learning for systems and systems for machine learning", in "Presentation at 2017 Conference on Neural Information Processing Systems", (2017).

Doerfler, D., J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. Malas, J.-L. Vay and H. Vincenti, "Applying the roofline performance model to the intel xeon phi knights landing processor", in "International Conference on High Performance Computing", pp. 339–353 (Springer, 2016).

Druzhkov, P. and V. Kustikova, "A survey of deep learning methods and software tools for image classification and object detection", Pattern Recognition and Image Analysis **26**, 1, 9–15 (2016).

Duan, B., W. Wang, X. Li, C. Zhang, P. Zhang and N. Sun, "Floating-point mixed-radix fft core generation for fpga and comparison with gpu and cpu", in "2011 International Conference on Field-Programmable Technology", pp. 1–6 (IEEE, 2011).

Duff, I. S., "Computer solution of large sparse positive definite systems (alan george and joseph w. liu", SIAM Review **26**, 2, 289–291 (1984).

Dusi, M., S. Napolitano, S. Niccolini and S. Longo, "A closer look at thin-client connections: Statistical application identification for qoe detection", IEEE Communications Magazine **50**, 11, 195–202 (2012).

Erdős, P. and A. Rényi, "On the evolution of random graphs", Publ. Math. Inst. Hung. Acad. Sci **5**, 1, 17–60 (1960).

Eyraud-Dubois, L., L. Marchal, O. Sinnen and F. Vivien, "Parallel scheduling of task trees with limited memory", ACM Transactions on Parallel Computing (TOPC) **2**, 2, 1–37 (2015).

Farahani, S., *ZigBee wireless networks and transceivers* (Newnes, 2011).

Feurer, M., A. Klein, K. Eggensperger, J. Springenberg, M. Blum and F. Hutter, "Efficient and robust automated machine learning", in "Advances in neural information processing systems", pp. 2962–2970 (2015).

Fiduccia, C. M. and R. M. Mattheyses, "A linear-time heuristic for improving network partitions", in "19th design automation conference", pp. 175–181 (IEEE, 1982).

Firasta, N., M. Buxton, P. Jinbo, K. Nasri and S. Kuo, "Intel AVX: New frontiers in performance improvements and energy efficiency", Intel white paper **19**, 20 (2008).

Ford, L. R. and D. R. Fulkerson, "Maximal flow through a network", Canadian journal of Mathematics **8**, 399–404 (1956).

Fowers, J., G. Brown, P. Cooke and G. Stitt, "A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications", in "Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays", pp. 47–56 (ACM, 2012).

Fowers, J., K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A configurable cloud-scale dnn processor for real-time ai", in "2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)", pp. 1–14 (IEEE, 2018).

Freitas, A. A. and S. H. Lavington, *Mining very large databases with parallel processing*, vol. 9 (Springer Science & Business Media, 1997).

Freitas, A. A. and S. H. Lavington, "Basic concepts on parallel processing", in "Mining Very Large Databases with Parallel Processing", pp. 61–69 (Springer, 2000).

Gajski, D. D., N. D. Dutt, A. C. Wu and S. Y. Lin, *High—Level Synthesis: Introduction to Chip and System Design* (Springer Science & Business Media, 2012).

Garey, M. R., D. S. Johnson and L. Stockmeyer, "Some simplified np-complete problems", in "Proceedings of the sixth annual ACM symposium on Theory of computing", pp. 47–63 (1974).

Glorot, X. and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", in "Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics", pp. 249–256 (2010).

Grady, L. and E. L. Schwartz, "Isoperimetric graph partitioning for image segmentation", IEEE transactions on pattern analysis and machine intelligence **28**, 3, 469–475 (2006).

Grosser, T., A. Groesslinger and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation", Parallel Processing Letters **22**, 04, 1250010 (2012).

Grosser, T., H. Zheng, R. Aloor, A. Simbürger, A. Größlinger and L.-N. Pouchet, "Polly-polyhedral optimization in llvm", in "Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)", vol. 2011, p. 1 (2011).

Guennebaud, G., B. Jacob *et al.*, "Eigen", URl: http://eigen. tuxfamily. org (2010).

Haartsen, J. C., "Bluetooth radio system", Wiley Encyclopedia of Telecommunications (2003).

Hager, W. W., D. T. Phan and H. Zhang, "An exact algorithm for graph partitioning", Mathematical Programming **137**, 1-2, 531–556 (2013).

Haney, R., T. Meuse, J. Kepner and J. Lebak, "The HPEC challenge benchmark suite", in "HPEC 2005 Workshop", (2005).

He, J., A. E. Snavely, R. F. Van der Wijngaart and M. A. Frumkin, "Automatic recognition of performance idioms in scientific applications", in "Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International", pp. 118–127 (IEEE, 2011).

He, K., X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition", in "Proceedings of the IEEE conference on computer vision and pattern recognition", pp. 770–778 (2016).

He, Y., J. Lin, Z. Liu, H. Wang, L.-J. Li and S. Han, "Amc: Automl for model compression and acceleration on mobile devices", in "Proceedings of the European Conference on Computer Vision (ECCV)", pp. 784–800 (2018).

Hegde, K., R. Agrawal, Y. Yao and C. W. Fletcher, "Morph: Flexible acceleration for 3d cnn-based video understanding", in "2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)", pp. 933–946 (IEEE, 2018).

Hendrickson, B. and R. W. Leland, "A multi-level algorithm for partitioning graphs.", SC **95**, 28, 1–14 (1995).

Henning, J. L., "SPEC CPU2000: Measuring CPU performance in the new millennium", Computer **33**, 7, 28–35 (2000).

Herdman, J., W. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. Mallinson and S. A. Jarvis, "Accelerating hydrocodes with OpenACC, OpenCL and CUDA", in "2012 SC Companion: High Performance Computing, Networking Storage and Analysis", pp. 465–471 (IEEE, 2012).

HPE, "HPE Edge Line EL1000", `https://www.hpe.com/` (2019a).

HPE, "HPE Edge Line EL4000", `https://www.hpe.com/` (2019b).

HPE, "HPE GL10 IoT Gateway", `https://www.hpe.com/` (2019c).

Hsu, R.-H., J. Lee, T. Q. Quek and J.-C. Chen, "Reconfigurable security: Edge-computing-based framework for iot", IEEE Network **32**, 5, 92–99 (2018).

Huang, G., Z. Liu, L. Van Der Maaten and K. Q. Weinberger, "Densely connected convolutional networks", in "Proceedings of the IEEE conference on computer vision and pattern recognition", pp. 4700–4708 (2017).

Hunkeler, U., H. L. Truong and A. Stanford-Clark, "Mqtt-s—a publish/subscribe protocol for wireless sensor networks", in "2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)", pp. 791–798 (IEEE, 2008).

Hyafil, L. and R. L. Rivest, *Graph partitioning and constructing optimal decision trees are polynomial complete problems* (IRIA. Laboratoire de Recherche en Informatique et Automatique, 1973).

Intel, "Fog Reference Unit", `https://www.intel.com/` (2017).

Intel, FPGA, "SDK for OpenCL, Programming guide (2017)", (2017).

Ji, S., W. Xu, M. Yang and K. Yu, "3d convolutional neural networks for human action recognition", IEEE transactions on pattern analysis and machine intelligence **35**, 1, 221–231 (2012).

Jia, H., Y. Zhang, G. Long, J. Xu, S. Yan and Y. Li, "GPURoofline: a model for guiding performance optimizations on GPUs", in "European Conference on Parallel Processing", pp. 920–932 (Springer, 2012).

Jia, Y., E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding", in "Proceedings of the 22nd ACM international conference on Multimedia", pp. 675–678 (ACM, 2014).

Jia, Z., J. Thomas, T. Warszawski, M. Gao, M. Zaharia and A. Aiken, "Optimizing dnn computation with relaxed graph substitutions", SysML 2019 (2019).

Jiang, W., E. H.-M. Sha, X. Zhang, L. Yang, Q. Zhuge, Y. Shi and J. Hu, "Achieving super-linear speedup across multi-fpga for real-time dnn inference", ACM Transactions on Embedded Computing Systems (TECS) **18**, 5s, 1–23 (2019).

Johnston, C., K. Gribbon and D. Bailey, "Implementing image processing algorithms on fpgas", in "Proceedings of the Eleventh Electronics New Zealand Conference, ENZCon'04", pp. 118–123 (2004).

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

Juega, J. C., J. I. Gómez, C. Tenllado and F. Catthoor, "Adaptive mapping and parameter selection scheme to improve automatic code generation for gpus", in "Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization", pp. 251–261 (2014).

Kahn, A. B., "Topological sorting of large networks", Communications of the ACM **5**, 11, 558–562 (1962).

Kahng, A. B., J. Lienig, I. L. Markov and J. Hu, *VLSI physical design: from graph partitioning to timing closure* (Springer Science & Business Media, 2011).

Kalarot, R. and J. Morris, "Comparison of fpga and gpu implementations of real-time stereo vision", in "2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition-Workshops", pp. 9–15 (IEEE, 2010).

Karypis, G. and V. Kumar, "Analysis of multilevel graph partitioning", in "Supercomputing'95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing", pp. 29–29 (IEEE, 1995).

Karypis, G. and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs", SIAM Journal on scientific Computing **20**, 1, 359–392 (1998a).

Karypis, G. and V. Kumar, "Multilevelk-way partitioning scheme for irregular graphs", Journal of Parallel and Distributed computing **48**, 1, 96–129 (1998b).

Kernighan, B. W. and S. Lin, "An efficient heuristic procedure for partitioning graphs", The Bell system technical journal **49**, 2, 291–307 (1970).

Kieritz, T., D. Luxen, P. Sanders and C. Vetter, "Distributed time-dependent contraction hierarchies", in "International Symposium on Experimental Algorithms", pp. 83–93 (Springer, 2010).

Koch, D. and J. Torresen, "Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting", in "Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays", pp. 45–54 (ACM, 2011).

Konstantinidis, A., P. H. Kelly, J. Ramanujam and P. Sadayappan, "Parametric gpu code generation for affine loop programs", in "International Workshop on Languages and Compilers for Parallel Computing", pp. 136–151 (Springer, 2013).

Kumar, S., V. Srinivasan, A. Sharifian, N. Sumner and A. Shriraman, "Peruse and profit: Estimating the accelerability of loops", in "Proceedings of the 2016 International Conference on Supercomputing", p. 21 (ACM, 2016).

Lamport, L., "The parallel execution of do loops", Communications of the ACM **17**, 2, 83–93 (1974).

Land, A. H. and A. G. Doig, "An automatic method for solving discrete programming problems", in "50 Years of Integer Programming 1958-2008", pp. 105–132 (Springer, 2010).

Lang, K. and S. Rao, "A flow-based method for improving the expansion or conductance of graph cuts", in "International Conference on Integer Programming and Combinatorial Optimization", pp. 325–337 (Springer, 2004).

Laredo, D., Y. Qin, O. Schütze and J.-Q. Sun, "Automatic model selection for neural networks", arXiv preprint arXiv:1905.06010 (2019).

Lavin, A. and S. Gray, "Fast algorithms for convolutional neural networks", in "Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition", pp. 4013–4021 (2016).

LeCun, Y., Y. Bengio and G. Hinton, "Deep learning", nature **521**, 7553, 436 (2015).

Lew, A. and H. Mauch, *Dynamic programming: A computational tool*, vol. 38 (Springer, 2006).

Ling, A. C., U. Aydonat, S. O'Connell, D. Capalija and G. R. Chiu, "Creating high performance applications with intel's fpga opencl™ sdk", in "Proceedings of the 5th International Workshop on OpenCL", pp. 1–1 (2017).

Litjens, G., T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken and C. I. Sánchez, "A survey on deep learning in medical image analysis", Medical image analysis **42**, 60–88 (2017).

Liu, H., K. Simonyan and Y. Yang, "Darts: Differentiable architecture search", arXiv preprint arXiv:1806.09055 (2018a).

Liu, L., W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu and M. Pietikäinen, "Deep learning for generic object detection: A survey", arXiv preprint arXiv:1809.02165 (2018b).

Liu, Z., P. Chow, J. Xu, J. Jiang, Y. Dou and J. Zhou, "A uniform architecture design for accelerating 2d and 3d cnns on fpgas", Electronics **8**, 1, 65 (2019).

Lo, Y. J., S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall and L. Oliker, "Roofline model toolkit: A practical tool for architectural and program analysis", in "International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems", pp. 129–148 (Springer, 2014).

Loechner, V., "Polylib: A library for manipulating parameterized polyhedra", (1999).

Luxen, D. and D. Schieferdecker, "Candidate sets for alternative routes in road networks", in "International Symposium on Experimental Algorithms", pp. 260–270 (Springer, 2012).

Ma, Y., Y. Cao, S. Vrudhula and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks", in "Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays", pp. 45–54 (ACM, 2017a).

Ma, Y., Y. Cao, S. Vrudhula and J.-s. Seo, "Optimizing the convolution operation to accelerate deep neural networks on fpga", IEEE Transactions on Very Large Scale Integration (VLSI) Systems **26**, 7, 1354–1367 (2018).

Ma, Y., M. Kim, Y. Cao, S. Vrudhula and J.-s. Seo, "End-to-end scalable fpga accelerator for deep residual networks", in "2017 IEEE International Symposium on Circuits and Systems (ISCAS)", pp. 1–4 (IEEE, 2017b).

Maleki, S., Y. Gao, M. J. Garzar, T. Wong, D. A. Padua *et al.*, "An evaluation of vectorizing compilers", in "Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on", pp. 372–382 (IEEE, 2011).

Marchal, L., B. Simon and F. Vivien, "Limiting the memory footprint when dynamically scheduling dags on shared-memory platforms", Journal of Parallel and Distributed Computing **128**, 30–42 (2019).

Maturana, D. and S. Scherer, "Voxnet: A 3d convolutional neural network for real-time object recognition", in "2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)", pp. 922–928 (IEEE, 2015).

Maue, J., P. Sanders and D. Matijevic, "Goal-directed shortest-path queries using precomputed cluster distances", Journal of Experimental Algorithmics (JEA) **14**, 3–2 (2010).

Meng, J., V. A. Morozov, K. Kumaran, V. Vishwanath and T. D. Uram, "GROPHECY: GPU performance projection from CPU code skeletons", in "Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis", p. 14 (ACM, 2011).

Meswani, M. R., L. Carrington, D. Unat, A. Snavely, S. Baden and S. Poole, "Modeling and predicting performance of high performance computing applications on hardware accelerators", The International Journal of High Performance Computing Applications **27**, 2, 89–108 (2013).

Moore, A., "Fpgas for dummies", Altera Special Edition. Hoboken: John Wiley & Sons, Inc (2014).

Moreira, O., M. Popp and C. Schulz, "Graph partitioning with acyclicity constraints", arXiv preprint arXiv:1704.00705 (2017).

Moreira, O., M. Popp and C. Schulz, "Evolutionary multi-level acyclic graph partitioning", Journal of Heuristics **26**, 5, 771–799 (2020).

Munshi, A., "The opencl specification", in "2009 IEEE Hot Chips 21 Symposium (HCS)", pp. 1–314 (IEEE, 2009).

Ndu, G., J. Navaridas and M. Luján, "CHO: towards a benchmark suite for OpenCL FPGA accelerators", in "Proceedings of the 3rd International Workshop on OpenCL", p. 10 (ACM, 2015).

Needleman, S. B. and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins", Journal of molecular biology **48**, 3, 443–453 (1970).

Nickolls, J. and W. J. Dally, "The GPU computing era", IEEE micro **30**, 2 (2010).

Nugteren, C., "Clblast: A tuned opencl BLAS library", arXiv preprint arXiv:1705.05249 (2017).

Nvidia, C., "Cublas library", NVIDIA Corporation, Santa Clara, California **15**, 27, 31 (2008).

Owens, J. D., M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, "Gpu computing", (2008).

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library", in "Advances in neural information processing systems", pp. 8026–8037 (2019).

Peng, B., L. Zhang and D. Zhang, "A survey of graph theoretical approaches to image segmentation", Pattern recognition **46**, 3, 1020–1038 (2013).

Premsankar, G., M. Di Francesco and T. Taleb, "Edge computing for the internet of things: A case study", IEEE Internet of Things Journal **5**, 2, 1275–1284 (2018).

Pytorch, "Pytorch", `https://pytorch.org` (????).

Qin, S. and M. Berekovic, "A comparison of high-level design tools for soc-fpga on disparity map calculation example", arXiv preprint arXiv:1509.00036 (2015).

Qiu, J., J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network", in "Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays", pp. 26–35 (ACM, 2016).

Real, E., A. Aggarwal, Y. Huang and Q. V. Le, "Regularized evolution for image classifier architecture search", in "Proceedings of the aaai conference on artificial intelligence", vol. 33, pp. 4780–4789 (2019).

Rozenberg, G., *Handbook of graph grammars and computing by graph transformation*, vol. 1 (World scientific, 1997).

Salihoglu, S. and J. Widom, "Gps: A graph processing system", in "Proceedings of the 25th International Conference on Scientific and Statistical Database Management", pp. 1–12 (2013).

Sanders, J. and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming* (Addison-Wesley Professional, 2010).

Sandler, M., A. Howard, M. Zhu, A. Zhmoginov and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks", in "Proceedings of the IEEE conference on computer vision and pattern recognition", pp. 4510–4520 (2018).

Schreiber, R., J. J. Dongarra *et al.*, *Automatic blocking of nested loops* (Research Institute for Advanced Computer Science, NASA Ames Research Center, 1990).

Sensen, N., "Lower bounds and exact algorithms for the graph partitioning problem using multicommodity flows", in "European Symposium on Algorithms", pp. 391–403 (Springer, 2001).

Settle, S. O. *et al.*, "High-performance dynamic programming on fpgas with opencl", in "Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)", pp. 1–6 (2013).

Sharma, H., J. Park, E. Amaro, B. Thwaites, P. Kotha, A. Gupta, J. K. Kim, A. Mishra and H. Esmaeilzadeh, "Dnnweaver: From high-level deep network models to fpga acceleration", in "the Workshop on Cognitive Architectures", (2016).

Shen, J., Y. Huang, Z. Wang, Y. Qiao, M. Wen and C. Zhang, "Towards a uniform template-based architecture for accelerating 2d and 3d cnns on fpga", in "Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays", pp. 97–106 (ACM, 2018).

Simbürger, A., S. Apel, A. Größlinger and C. Lengauer, "The potential of polyhedral optimization: An empirical study", in "2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)", pp. 508–518 (IEEE, 2013).

Simon, H. D. and S.-H. Teng, "How good is recursive bisection?", SIAM Journal on Scientific Computing **18**, 5, 1436–1445 (1997).

Singh, D. and S. P. Engineer, "Higher level programming abstractions for fpgas using opencl", in "Workshop on Design Methods and Tools for FPGA-Based Acceleration of Scientific Computing", (2011).

Skorin-Kapov, L. and M. Matijasevic, "Analysis of qos requirements for e-health services and mapping to evolved packet system qos classes", International journal of telemedicine and applications **2010**, 9 (2010).

Stanton, I. and G. Kliot, "Streaming graph partitioning for large distributed graphs", in "Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining", pp. 1222–1230 (2012).

Suda, N., V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks", in "Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays", pp. 16–25 (ACM, 2016).

Sun, L., K. Jia, D.-Y. Yeung and B. E. Shi, "Human action recognition using factorized spatio-temporal convolutional networks", in "Proceedings of the IEEE international conference on computer vision", pp. 4597–4605 (2015).

Sze, V., Y.-H. Chen, T.-J. Yang and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey", Proceedings of the IEEE **105**, 12, 2295–2329 (2017).

Tran, D., L. Bourdev, R. Fergus, L. Torresani and M. Paluri, "Learning spatiotemporal features with 3d convolutional networks", in "Proceedings of the IEEE international conference on computer vision", pp. 4489–4497 (2015).

Trifunovic, K., A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin and R. Upadrasta, "Graphite two years after: First lessons learned from real-world polyhedral compilation", in "GCC Research Opportunities Workshop (GROW'10)", (2010).

Tsourakakis, C., C. Gkantsidis, B. Radunovic and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs", in "Proceedings of the 7th ACM international conference on Web search and data mining", pp. 333–342 (2014).

Vasilache, N., O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions", arXiv preprint arXiv:1802.04730 (2018).

Vinoski, S., "Advanced message queuing protocol", IEEE Internet Computing , 6, 87–89 (2006).

Vissers, K., "Versal: The xilinx adaptive compute acceleration platform (acap)", in "Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays", pp. 83–83 (ACM, 2019).

Waidyasooriya, H. M. and M. Hariyama, "Multi-fpga accelerator architecture for stencil computation exploiting spacial and temporal scalability", IEEE Access (2019).

Wang, D., K. Xu and D. Jiang, "Pipecnn: An opencl-based open-source fpga accelerator for convolution neural networks", in "2017 International Conference on Field Programmable Technology (ICFPT)", pp. 279–282 (IEEE, 2017).

Wang, J., L. Guo and J. Cong, "Autosa: A polyhedral compiler for high-performance systolic arrays on fpga", in "Proceedings of the 2021 ACM/SIGDA international symposium on Field-programmable gate arrays", (2021).

Wang, K., Z. Liu, Y. Lin, J. Lin and S. Han, "Haq: Hardware-aware automated quantization with mixed precision", in "Proceedings of the IEEE conference on computer vision and pattern recognition", pp. 8612–8620 (2019a).

Wang, L. and D. Sng, "Deep learning algorithms with applications to video analytics for a smart city: A survey", arXiv preprint arXiv:1512.03131 (2015).

Wang, Y., S. Wang, M. Zhou, Q. Jiang and Z. Tian, "Ts-i3d based hand gesture recognition method with radar sensor", IEEE Access **7**, 22902–22913 (2019b).

Watts, D. J. and S. H. Strogatz, "Collective dynamics of 'small-world'networks", nature **393**, 6684, 440–442 (1998).

Williams, S., A. Waterman and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures", Communications of the ACM **52**, 4, 65–76 (2009).

Wilson, P., *Design Recipes for FPGAs: Using Verilog and VHDL* (Newnes, 2015).

Winkler, T. and B. Rinner, "Security and privacy protection in visual sensor networks: A survey", ACM Computing Surveys (CSUR) **47**, 1, 2 (2014).

Winograd, S., "On multiplication of polynomials modulo a polynomial", SIAM Journal on Computing **9**, 2, 225–229 (1980).

Wortsman, M., A. Farhadi and M. Rastegari, "Discovering neural wirings", in "Advances in Neural Information Processing Systems", pp. 2684–2694 (2019).

Xie, S., A. Kirillov, R. Girshick and K. He, "Exploring randomly wired neural networks for image recognition", in "Proceedings of the IEEE International Conference on Computer Vision", pp. 1284–1293 (2019).

Yazdanbakhsh, A., A. T. Elthakeb, P. Pilligundla, F. Mireshghallah and H. Esmaeilzadeh, "Releq: A reinforcement learning approach for deep quantization of neural networks", (2018).

Yi, S., C. Li and Q. Li, "A survey of fog computing: concepts, applications and issues", in "Proceedings of the 2015 Workshop on Mobile Big Data", pp. 37–42 (ACM, 2015).

Zaharia, M., R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: a unified engine for big data processing", Communications of the ACM **59**, 11, 56–65 (2016).

Zhang, C., 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", pp. 161–170 (ACM, 2015).

Zhang, C., G. Sun, Z. Fang, P. Zhou, P. Pan and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2018).

Zhang, C., D. Wu, J. Sun, G. Sun, G. Luo and J. Cong, "Energy-efficient cnn implementation on a deeply pipelined fpga cluster", in "Proceedings of the 2016 International Symposium on Low Power Electronics and Design", pp. 326–331 (ACM, 2016).

Zhang, Y., Y. H. Shalabi, R. Jain, K. K. Nagar and J. D. Bakos, "Fpga vs. gpu for sparse matrix vector multiply", in "2009 International Conference on Field-Programmable Technology", pp. 255–262 (IEEE, 2009).

Zohouri, H. R., N. Maruyama, A. Smith, M. Matsuda and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with fpgas", in "SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis", pp. 409–420 (IEEE, 2016).

Zohouri, H. R., A. Podobas and S. Matsuoka, "Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL", in "Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays", pp. 153–162 (ACM, 2018).

Zoph, B. and Q. V. Le, "Neural architecture search with reinforcement learning", arXiv preprint arXiv:1611.01578 (2016).

Zoph, B., V. Vasudevan, J. Shlens and Q. V. Le, "Learning transferable architectures for scalable image recognition", in "Proceedings of the IEEE conference on computer vision and pattern recognition", pp. 8697–8710 (2018).