

AMERICAN UNIVERSITY OF BEIRUT

DISTRIBUTED FPGA-BASED
ACCELERATION OF BIG DATA ANALYTICS
IN THE DATA CENTER ENVIRONMENT

by

RAGHID HIKMAT MORCEL

A dissertation
submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
to the Department of Electrical and Computer Engineering
of the Maroun Semaan Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
September 2018

AMERICAN UNIVERSITY OF BEIRUT

DISTRIBUTED FPGA-BASED
ACCELERATION OF BIG DATA ANALYTICS
IN THE DATA CENTER ENVIRONMENT

by
RAGHID HIKMAT MORCEL

Approved by:


Dr. Hassan Artail, Professor
Electrical and Computer Engineering

Committee Chair



Dr. Haitham Akkary, Professor
Electrical and Computer Engineering

Advisor

 for Dr. Akkary

Dr. Hazem Hajj, Professor
Electrical and Computer Engineering

Member of Committee



Dr. Mazen A. R. Saghir, Professor
Electrical and Computer Engineering

Member of Committee



Dr. Abbas Amira, Professor

Member of Committee

Qatar University

 for Dr. Amira

Dr. Yasser Mohanna, Professor

Member of Committee

Lebanese University



Date of thesis defense: September 3, 2018

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT
RELEASE FORM

Student Name: Horzel Raghd Hikmat
Last First Middle

Master's Thesis Master's Project Doctoral Dissertation

I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after: **One ___ year from the date of submission of my thesis, dissertation or project.**
Two ___ years from the date of submission of my thesis, dissertation or project.
Three ✓ years from the date of submission of my thesis, dissertation or project.

Raghd February 6, 2019
Signature Date

This form is signed when submitting the thesis, dissertation, or project to the University Libraries

Contents

Abstract	1
1 Introduction	6
1.1 Motivation	6
1.2 Problem Statement	11
1.3 Proposed Solutions and Contributions	13
1.3.1 A scalable Network-attached deployment model for FPGAs.	14
1.3.2 A design methodology for mapping ConvNet inference work-loads to FPGA accelerators.	18
1.4 Thesis Outline	19
2 Background for Data Centers and Big Data Architectures	21
2.1 Data Center Architectures	21
2.1.1 Data Center Design Factors	22
2.1.2 Data Center Network Infrastructure	23
2.1.3 Data Center design models	25
2.1.4 Data Center Scalability and Power Efficiency issues	29
2.2 Big Data Architectures	32
2.2.1 Stages of Big Data	32
2.2.2 Big Data management frameworks	35
3 Network-Attached Reconfigurable Accelerator System Architecture for the Spark Data Center	47
3.1 FPGA-based Deployment Models	50
3.2 Related Work	52
3.2.1 Co-processor implementations	52
3.2.2 Network-Attached Accelerator implementations	58
3.3 Proposed Deployment Model	61
3.3.1 Challenges	61
3.3.2 Network-Attached Accelerator system architecture	62
3.3.3 NAA Node Architecture	64
3.3.4 NAA Compute Model	66
3.3.5 Firmware Architecture	72

3.4	Experimental Evaluation	81
3.4.1	NAA Platform	81
3.4.2	Firmware Implementation	82
3.4.3	Experimental Setup	82
3.5	Conclusion	83
4	Accelerating Convolutional Neural Network Operations in the Spark Data Center Environment	85
4.1	The Multi-layer Convolution Operation	88
4.2	Related Work	90
4.3	Fpga-based accelerator for convolutional networks	94
4.3.1	System Overview	94
4.3.2	Hardware Architecture	95
4.3.3	Software Layer	100
4.4	Methodology and experimental results	101
4.4.1	Experimental Setup	101
4.4.2	Benchmark	101
4.4.3	Speedups resulting from employing the FPGA-based accelerator	102
4.4.4	Performance Model for the Multi-layer Convolution Operation	105
4.4.5	Energy Saving Resulting from employing the FPGA-base accelerator	108
4.5	Conclusion	110
5	FeatherNet: An Accelerated Convolutional Neural Network Design for Resource-Constrained FPGAs	111
5.1	Deep Convolutional Neural Networks	116
5.2	Related Work	119
5.3	Design Methodolgy	124
5.3.1	Design Challenges	124
5.3.2	Graphical Representation and Modeling of Neural Inference computation	126
5.3.3	Optimization and Reduction schemes for 2D Convolution	135
5.3.4	Optimizing for Finite Word-length Representation and Computation	151
5.4	Minimalists Accelerated Convnet System Architecture	155
5.4.1	Proposed Architecture	156
5.4.2	Modularity and Portability	160
5.4.3	Design Entry and Implementation	162
5.5	Evaluation and Results	165
5.5.1	Evaluation and Experimental Setup	165
5.5.2	Results	166

5.6	Conclusion	171
6	Conclusions and Future Perspectives	172
6.1	Conclusions	172
6.2	Future Perspectives	176
6.2.1	Network Attached Accelerator Optimizations	176
6.2.2	Optimizing FeatherNet for Performance	178
A	Proof for the Proposed Transformation in Figure 5.7	182
	Bibliography	184

List of Figures

1.1	Digital Universe size predictions for 2020	8
1.2	Deploying FPGAs in data center facilities	15
1.3	A typical scenario where several FPGA boards are allocated to two different applications	16
2.1	Cisco's Basic Layered Design	24
2.2	A logic view of a server cluster	27
2.3	A physical view of a server cluster	28
2.4	HPC Architectures Shift Toward Cluster Computing	29
2.5	HDFS distributing data blocks to rack servers	36
2.6	two client processes submitting two different MapReduce Jobs to a Hadoop Resource Manager	39
2.7	Hadoop MapReduce DataFlow	40
2.8	Logical view of a Spark cluster	43
3.1	Catapult's 6×8 torus network topology and the physical wiring on a pod of servers.	53
3.2	Components of the Shell Architecture.	54
3.3	Intel's Xeon+FPGA platform.	55
3.4	A Snippet Blade.	56
3.5	The SDAccel Development Environment.	57
3.6	The RAD architecture.	59
3.7	RASSD node prototype.	60
3.8	NAA system architecture	63
3.9	NAA system logical view	64
3.10	NAA node Architecture.	65
3.11	Accelerator's interfaces: parameters, input and output interfaces.	68
3.12	NAA computational pipeline example.	70
3.13	Layered Architecture	74
3.14	Integrating our NAA system architecture into Spark	80
3.15	Experimental Setup	83
4.1	System Overview.	95
4.2	Functional Architecture of our compute node.	96

4.3	Optimized architecture of the multi-layer convolution.	98
4.4	Optimized 1D-FIR filter.	98
4.5	Speedup over an ARM core implementation as a function of the number of input feature maps	103
4.6	Speedup over a Core i7 CPU core implementation as a function of the number of input feature maps	104
4.7	The achievable speedup as function of the operating frequency. . .	105
4.8	Latency of computing a single output feature map on the FPGA and the slope	107
5.1	An illustration of a typical Convolutional Layer.	118
5.2	Block diagram representation for the 1-D FIR filter (a) and for the 2-D FIR (b)	129
5.3	A hardware realization for the 2-D convolution filter derive from figure 5.2(b).	130
5.4	A block diagram representation for a Local response normalization filter with $n = 3$	133
5.5	A block diagram representation for a 2-D Pooling filter with kernel size 3×3 and a stride of 1 pixel	135
5.6	A symbolic notation for the compressor and a typical example with $M = 2$	137
5.7	Multi-rate signal processing transformations	137
5.8	A block diagram representation for 1-D convolution with a stride of 2 samples	138
5.9	Partially transformed 1-D convolution with a stride of 2	140
5.10	Fully transformed 1-D convolution	140
5.11	A minimal hardware implementation for a 1-D convolution filter with a size of 3 and a stride of 2	141
5.12	A minimal hardware implementation for a 2-D convolution filter with a size of 3×3 and a stride of 2	142
5.13	Coefficient Scrambler object feeding a 1D- filter with a mask size 3, stride 2, and padding 1	144
5.14	A 2D- Coefficient Scrambler object feeding a 2D- filter with a mask size 3×3 , stride 2, and padding 1	146
5.15	A 2D- Coefficient Scrambler object	147
5.16	A 2D- FIR filter with mask size 3×3 and stride 2	149
5.17	The sample and line accumulation grids that are common among all 2D-fir filters	149
5.18	Case where the outputs of two 2D-convolution filters are combined into a single output feature	151
5.19	Two 2D-FIR filters sharing the same sample and line accumulate networks	152

5.20	The effect of fixed-point representation of network parameters on the accuracy of AlexNet	153
5.21	System Architecture	157
5.22	The memory mux component	159
6.1	Modified NAA Node Architecture	178
A.1	T delay elements followed by an M -fold compressor	182
A.2	T delay elements rearranged into S delay element followed by $T-S$ delay elements	183

List of Tables

4.1	Resource Utilization.	102
5.1	AlexNet layers and their hyper parameters.	119
5.2	Hardware cost of different 2-D FIR filters	131
5.3	Hardware cost of different 2-D FIR filters with strides	148
5.4	Possible fixed-point representation for every layer in AlexNet . . .	154
5.5	Evaluation Platforms and their characteristics	165
5.6	Resource Utilization for AlexNet on ZedBoard and Cyclone V . .	167
5.7	Performance and Energy Results of AlexNet implementation with- out the on-chip cache.	167
5.8	Latency of individual compute stages of AlexNet with the on-chip cache included.	168
5.9	Comparison between our implementation and other work in the literature.	169

An Abstract of the Dissertation of

Raghid Hikmat Morcel for Doctor of Philosophy
Major: Electrical and Computer Engineering

Title: Distributed FPGA-based acceleration of Big Data analytics
in the Data Center Environment.

Several research studies have shown that Big data, as a largescale phenomenon, is creating substantial value for the world's economy by boosting the productivity and competitiveness of private-sector businesses and public enterprises, and thus it is generating extensive economic surplus for consumers. However, the computational limitations of general-purpose processor-based data centers are preventing businesses from fully integrating Big data architectures in their business model. A widely accepted solution to this problem consists of supplementing microprocessors with application-specific hardware accelerators. Although a wide-ranging literature exists on the benefits of employing Field Programmable Gate Arrays (FPGAs) as hardware accelerators in the data center environment, FPGAs have not been extensively deployed there for two reasons: first, the lack of a scalable and power-efficient data center deployment model for FPGAs, and second, the complexity of programming and developing FPGA-based hardware accelerators for data center workloads.

In this thesis, we address two problems: (1) the scalable deployment of FPGA platforms, and (2) the design and implementation of FPGA-based hardware accelerators for Deep Learning Big Data workloads. To address the first problem, a scalable and power-efficient Network-Attached-Accelerator (NAA) system architecture is proposed. The proposed NAA system architecture allows the seamless and efficient integration of FPGA-based platforms in existing data center architectures. Experimental Results showed that orders-of-magnitude improvements in performance can be achieved in certain applications. Then, to address the second problem, we propose a design methodology for mapping Deep Convolutional Neural Network (ConvNet) inference workloads to FPGA hardware accelerators. We employed our methodology to accelerate AlexNet, a popular computationally intensive ConvNet used for accurate image classification, and we showed that our design can achieve a potential performance of up to 9 frames/sec on a very resource-restricted FPGA platform versus 2.84 frames/sec on a Core i7 processor core.

Abbreviations

1RU	1 Rack Unit
ANSI	American National Standards Institute
API	Application Programming Interface
AM	Application Master
ASIC	Application Specific Integrated Circuit
ACB	Accelerator Control Block
BICSI	Building Industry Consulting Service International
BRAM	Block Random Access Memory
ConvNet	Convolutional Neural Network
CPU	Central Processing Unit
CRM	Customer Relationship Management
CNP	Convolutional Neural Network Processor
CNN	Convolutional Neural Network
CIFAR	Canadian Institute For Advanced Research
CDM	Coefficient Delivery Mold
DevKit	Development Kit
DSP	Digital Signal Processor
DAS	Direct-Attached Storage
DAG	Direct Acyclic Graph
DRAM	Dynamic Synchronous Random Access Memory
DMA	Direct Memory Access
DG	Directed Graph
DLA	Deep Learning Accelerator
ERP	Enterprise Resource Planning
ERR	Energy Reduction Ratio
FPGA	Field Programmable Gate Array
FIR	Finite Impulse Response Filter
GPU	Graphical Processing Unit
GPO	Giga Operations Per Second
HPC	High-Performance Computing
HDFS	Hadoop Distributed File System
HLS	High-Level Synthesis
IDC	International Data Corporation

IoT	Internet of Things
ILSVRC	Large Scale Visual Recognition Challenge
IT	Information Technology
IP	Internet Protocol
IDE	Integrated Development Environment
IRFoR	Internal Routing and Forwarding Offload Engine
JAR	Java ARchive
LUT	LookUp Table
LB	Logic Block
LRN	Local Response Normalization
MPP	Massively Parallel Processing
MWS	Middleware Server
MoC	Model of Computation
MNIST	Modified National Institute of Standards and Technology
NIC	Network Interface Card
NAA	Network-Attached Accelerator
NAS	Network-Attached Storage
NRDC	National Resource Defense Council
NN	Neural Network
OLS	Ordinary Least Squares
PCIe	Peripheral Component Interconnect express
PDMoC	Parametrized DataFlow Model of Computation
PCAP	Processor Configuration Access Port
PSGD	Parallel Stochastic Gradient Descent Algorithm
PE	Processing Element
QPI	Quick Path Interconnect
RDD	Resilient Distributed Dataset
RTL	Register Transfer Language
RAD	Reconfigurable Active Drive
RASSD	Reconfigurable Active Solid State Drive
SAN	Storage Area Network
SAS	Serial Attached SCSI
SMP	Symetric Multi-Processing
SQL	Structured Query Language
SSD	Solid State Drive
SoC	System On a Chip
SSE	Streaming Single-Instruction-Multiple-Data Extension
TIA	Telecommunications Infrastructure Standard Association
TCP	Transport Control Protocol
TCPoE	TCP Offload Engine
UCLA	University of California, Los Angeles
UDP	User Datagram Protocol
UAV	Unmanned Aerial Vehicles

VLAN	Virtual Local Area Network
VALU	Vector Arithmetic and Logic Unit
Yarn	Yet Another Resource Manager

Chapter 1

Introduction

1.1 Motivation

Big Data has been promoted as being the umbrella term for very large datasets whose size and complexity are beyond the ability of legacy hardware and software technologies to capture, store and process [1, 2]. Recently, the term Big data gained a lot of popularity, reflecting the exponentially expanding volume of the digital universe and the resulting new technological advancements developed to address the technical challenges of this explosive growth.

Examples of Big Data usages may be found in almost every sector in the modern economy. In the private sector, for instance, companies and enterprises, going about their daily business routines, collect and store massive amounts of data about their customers, their suppliers, and their various economic and financial operations. The captured data is, then, analyzed to extract valuable insights and to guide the business during its complex decision-making process. In the civil and construction engineering sector, hundreds or even thousands of sensors are embedded in buildings, bridges, dams, and other megastructures to proactively and predictively monitor their health and integrity, and thus ensuring the safety of those who utilize and depend on these structures [3]. In healthcare,

data collected from patients are used to forecast a medical emergency and treat it pro-actively before it incurs an irreversible damage; for example, the UCLA brain surgery department is using Big Data Analytic tool-kits developed by IBM to capture and analyze real-time data streams of vital signs collected from multiple bedside monitoring equipment. The collected data is used to create timely alarms intended to predict sudden changes in the brain pressure in patients with traumatic brain injuries. These well-timed alarms give nurses and physicians more time to prevent further brain damages and to potentially save thousands of lives [4]. The availability of cheap consumer devices such as smart phones, Personal Computers, and laptops, allowed a large portion of the world population to easily access the internet. Consequently, billions of people are now producing their own “trails of data” and enhancing their online presence, mainly through social media platforms and services such as Facebook, Twitter, and Google+ [1]. In summary, Big Data as a phenomenon, is creating substantial value for humanity to the point that it became one of the indispensable factors of production in the modern economy, without which advanced economic activities cannot even take place [1].

This data collection behavior, although it is beneficial for the economy, comes at a great price. The collected data must be stored and processed in consolidated data center facilities; and the sheer size and variety of this collected data can put a huge strain on the data center’s storage and computing infrastructures. Many studies suggest that the amount of data collected or generated by individuals and corporations combined has been rising rapidly and will continue to grow exponentially in the upcoming years. An IDC report, for instance, claims that the total volume of digital information created and replicated on the internet will grow to almost 44 trillion gigabytes by 2020 [5]. Figure 1.1 depicts the predicted

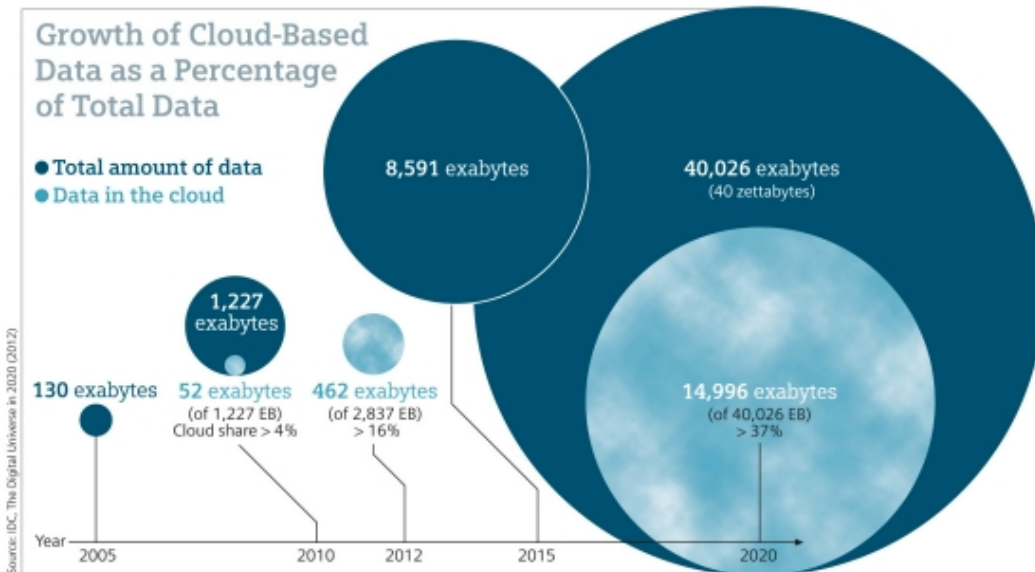


Figure 1.1: Digital Universe size predictions for 2020 [6]

size of the digital universe in 2020.

The sheer size of the collected data is not the only technical challenge of employing Big Data; the raw unstructured nature of this data, requires innovative algorithmic solutions and procedures that are usually computationally very intensive [1, 7]. Many recent procedures were developed to capture value and derive insights from unstructured Big Data; among those procedures is Deep Learning. Deep Learning is a blanket term for a set of algorithms and techniques that can automatically extract high-level, and complex abstract data representations from large volumes of unstructured data [7, 8]. The ability of Deep Learning algorithms to efficiently and accurately learn useful data representations makes them an indispensable tool-set in any Big Data Analytic toolkit. As a consequence, in the last couple of years, Deep Learning has been extensively used in many Big Data realms such as largescale computer vision and scene understanding [9–11].

One of the most prominent Deep Learning methods employs Deep Convo-

lutional Multi-Layer Neural Networks (ConvNets). Deep Convolutional Neural Networks (ConvNets) were inspired by the structure and function of the visual cortex in the human brain [12], and are typically employed in the object recognition/detection and in image classification. The main distinguishing feature of these networks is that they incorporate convolutional layers in their structure. Empirical studies have shown that Deep Convolutional Neural Networks outperform any form of shallow Neural Networks in many applications such as character and object recognition/detection and can even achieve human-like or even super-human performances in certain applications [8]. The only drawback of employing Deep ConvNets in Big Data Analytics is their tremendous computational complexity. For instance, some complex ConvNet architectures such as AlexNet [13] may take 20 to 30 hours to train. Generally, the bigger the Network, the more time it takes to train. ConvNet Inference is also computationally intensive; especially in certain Big Data scenarios where the inference task should run on a very large number of inputs.

A report on Big Data published by McKinsey & Company [1], a worldwide management consulting firm, pointed out that new innovations in software and hardware technologies are required, to capture the full potential of Big Data, since the limitations of legacy computing systems are preventing businesses from fully integrating Big Data architectures into their business models. Many software innovations that helped kick-start the use of Big Data in the enterprise, were made in the last decade. In 2004, Google conceived MapReduce [14], which is a parallel programming model and a distributed computing middleware used to process largescale datasets on distributed computing resources in a server cluster data center environment. MapReduce achieved tremendous success during its lifetime, and inspired a lot of other companies to develop similar Big Data

software frameworks. For Example, Hadoop MapReduce [15] is a free and open source implementation of the original Google MapReduce framework. Dryad [16] is another Big Data framework that provides a more comprehensive parallel programming model than MapReduce.

Although MapReduce gained a lot of popularity during the last decade, Many studies have shown that it was ill-suited for implementing applications that are iterative or interactive in nature [17,18]. Consequently, many alternative parallel frameworks were proposed to solve the issues of MapReduce. This led to a wide variety of specialized parallel frameworks such as Pregel [19], and GraphLab [20]. In 2010, however, Zaharia et al. [18] proposed a unified general-purpose in-memory Big Data computing framework called Spark. The Spark framework gained special attention and a lot of popularity due to its expressive power and very high performance. Nowadays, most of Big Data Analytic toolkits are built on top of the Spark framework [21].

Studies have shown that current established trends such as reliance on social media, search engines, smart phones, and IoT devices will further stimulate the exponential growth of the digital universe [1, 5, 22]. Moreover, application demands for computational performance have already outpaced the processor's maximum achievable computational capacity. To meet the ever-increasing demand for application performance, enterprises resort to scaling up their data centers' computational capacity by increasing the number processor cores. This strategy, however, leads to power-hungry facilities, as every additional processor core consumes a lot of power even when it is in an idle state. A Study has shown that, in 2013, data centers in the U.S. alone consumed 91 billion Kilowatt-hours of electricity, which is equivalent to twice the power consumption of all households in New York city [23]. In fact, the power consumption of data centers around

the world is becoming a significant environmental issue as their carbon impact is growing significantly higher every year.

This energy efficiency issue is common among all CPU-bound computing platforms. Recently various research endeavors have shown that employing Application Domain Specific computing platforms such as Graphical Processing Units (GPUs) and Field Programmable Gates arrays (FPGAs) may significantly benefit the data center environment in terms of power efficiency [24–26]. FPGAs, however, allow developing custom hardware accelerators that are fully adapted for certain applications and consequently they may provide better *power × performance* figures than GPUs [27]. Moreover, certain complex workloads such as deep Convolutional Neural Network inference fits very well into the micro-architecture of FPGA platforms, since FPGAs may be used to deploy customized reconfigurable data-paths that are optimized for convolutions.

To summarize, capturing the full benefits of Big Data requires innovations in both hardware and software technologies to take place. To keep up with the ever-increasing demand for application performance, FPGAs can be deployed in the data center environment to accelerate a wide-range of time-critical workloads, and Big data software frameworks such as Spark must be extended to allow it to target the deployed FPGAs.

1.2 Problem Statement

In this work, we aim at developing technologies and architectures that ease the deployment of FPGA-based hardware accelerators in the modern data center environment. Although the benefits of employing FPGAs to accelerate computationally intensive workloads are well known in the literature [24–26], FPGA

accelerators for Deep Learning workloads have not been widely deployed in the data center. We identify two main reasons: (1) FPGA deployment models and system architectures are not mature enough to allow the seamless integration of FPGA-based accelerators in a data center facility, and (2) the complexity of programming and developing FPGA-based accelerators for Deep Learning workloads, or Big Data Analytics in general, forms a true barrier that prevents FPGAs from gaining a lot of popularity among Big Data application developers. Typical services and applications that run in the data center environment are computationally complex and require deploying many FPGA devices to meet their demands [28]. Moreover, data center operators should be able to flexibly control the number and types of the deployed FPGA devices in a particular data center facility; for instance, a data center operator may suddenly decide to scale out its data center facility to meet seasonal surges in demand for computing power. It may also decide to replace several deployed high-end FPGA devices with low-end FPGA devices, to reduce power consumption; or it may do the exact opposite, to improve performance. Consequently, successfully deploying FPGA platforms in a data center environment requires (1) a scalable and (2) an energy efficient FPGA deployment model. Furthermore, in cases where data center operators decide to employ low-end FPGA devices in their facilities, a clever design methodology that allows fitting very complex Deep Learning workloads into low-end FPGA devices is also required. Finally, any FPGA system architecture should be able to seamlessly integrate into current Big Data management platforms such as Apache Hadoop [15] and Spark [18,29], thus these Big Data management platforms should be extended to provide support for FPGAs.

1.3 Proposed Solutions and Contributions

As mentioned earlier, successfully deploying FPGAs in the data center environment to accelerate a handful of distributed Big Data applications requires three essential components: (1) a scalable and flexible FPGA deployment method, (2) a method to seamlessly integrate FPGA devices in current Big Data management platforms (e.g. Hadoop and Spark), and (3) a library of FPGA configuration bit-streams that consists of custom-designed accelerators for a handful of desired applications. Note that addressing the second component, i.e., the seamless integration of FPGA devices in a Big Data management platform such as Spark, requires extending the platform with the capability to target the deployed FPGA devices. In this work, we address the first and third components only, i.e., the scalable FPGA deployment and the development of a library of custom-designed accelerators, and we leave extending the Spark platform to a future work.

In the first part of this work, we address the scalable and flexible deployment of FPGA devices; we accomplish this objective first by proposing a Network-attached accelerator deployment model for FPGAs in the data center environment, and second by developing a low-level firmware whose job is to mediate between the extended Spark middle-ware layer and the deployed FPGA devices. The proposed firmware should not be understood as an extension to the Spark environment to support FPGA acceleration from the Spark user perspective, but rather a low-level interface through which the extended middle-ware layer can program, configure, and launch computations on the deployed FPGA devices; as mentioned earlier, in this work, we do not attempt to design or implement this extension, instead we leave the design and implementation of an extended Spark middle-ware to a future work. In the second part of this work, we address

the design and implementation of FPGA-based accelerators (bitstreams) for deep Convolutional Neural Networks (ConvNet), in particular, we address the problem of fitting complex ConvNet architectures [13, 30–32] on resource-constrained FPGA devices [33–36]. We propose a novel DSP-based design methodology for mapping complex ConvNet inference sub-tasks to area, latency, and energy efficient FPGA realizations that can be implemented directly in FPGA logic. The resulting ConvNet accelerator architectures are suitable for deployment on low-end FPGA devices that can be installed in small data center facilities and in low-power IoT devices. In the following, we provide a brief overview of the proposed solutions i.e., the scalable Network-attached deployment model, and the design methodology for mapping complex ConvNet inference workloads to resource-constrained FPGA devices.

1.3.1 A scalable Network-attached deployment model for FPGAs.

The most common method of deploying FPGA-based accelerators in a traditional computer system consists of connecting FPGA daughtercards to computer nodes via PCIe edge connectors. For this reason, most of the previous attempts at deploying FPGA-based accelerators in the data center environment were largely based on PCIe-attached FPGA boards [28, 37, 38]. Figure 1.2(a) illustrates a server cluster data center with FPGA daughtercards connected to several server nodes via PCIe. With this deployment method, applications that target the FPGAs benefit from the considerably fast PCIe interconnect between the CPUs and the FPGA cards. A key disadvantage of this method, however, is that it doesn't scaleup efficiently. Recall that typical services and applications that

run in the data center environment are computationally complex and require deploying too many FPGAs to meet their demands. Introducing a new PCIe daughtercard with an FPGA device into a server cluster always requires a CPU-based server node to host it. Given that CPU cores consume significant amounts of power even when they are idle, scaling out the number of FPGA devices, with the PCIe deployment method, significantly increases the power consumption of the entire facility.

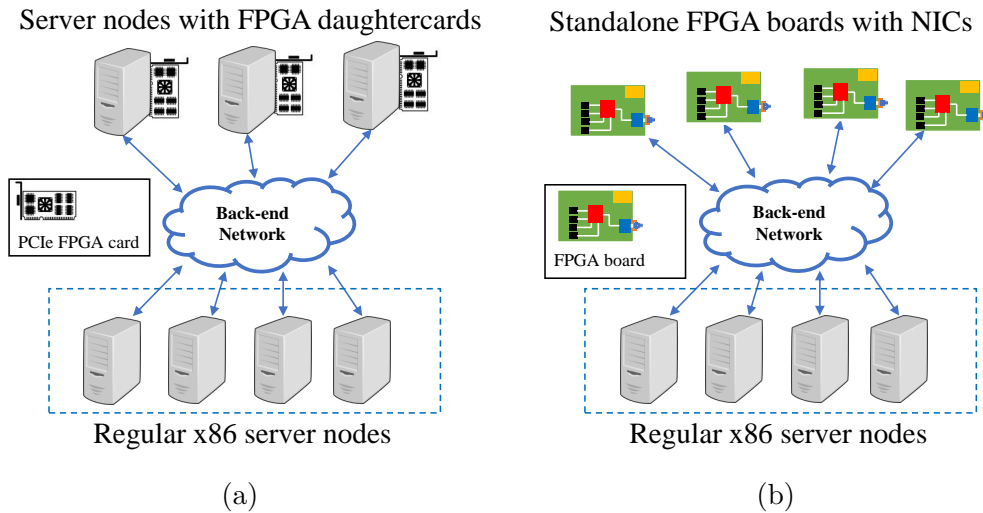


Figure 1.2: Deploying FPGAs in data center facilities

In this work, we propose a different approach for deploying FPGA devices in the server cluster data center founded on the Network-Attached Accelerator (NAA) deployment model. In this deployment method, every FPGA board in the data center has an Ethernet Network Interface Card (NIC) through which the board can be attached directly to the network infrastructure. Figure 1.2(b) illustrates a data center augmented with Network-Attached FPGA Boards. In addition to the FPGA device and the Ethernet interface, every FPGA board contains several on-board DRAM memory chips for local caching and storage. The FPGA device can implement an arbitrary custom-designed accelerator for

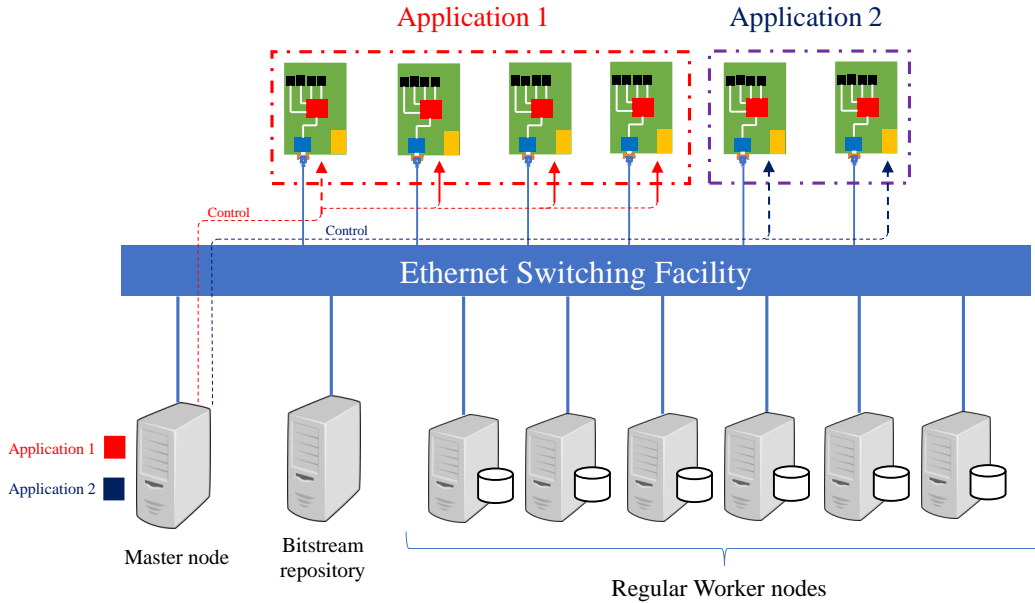


Figure 1.3: A typical scenario where several FPGA boards are allocated to two different applications

an arbitrary computation.

Figure 1.3 illustrates a typical scenario where several FPGA boards are allocated to two different applications. In order to run a compute job on the allocated FPGA boards, the Master node (c.f. figure 1.3) contacts the Bitstream repository to fetch suitable FPGA bitstreams for the applications. It reconfigures the allocated FPGA devices using the fetched bitstreams, and then sends coarse-grained task descriptions to each individual FPGA board. When the FPGA boards receive their task descriptions, they execute those tasks in parallel on the FPGA accelerators. The worker nodes may as well receive task descriptions that instruct each worker node to send certain local data partitions to a specific FPGA board. In addition to the Network-attached deployment model, we also devise a system architecture and a firmware design that allow the master node to allocate, program, and launch computational tasks on the FPGA boards. This work includes several contributions:

- A method for deploying FPGA devices directly into the data center’s network infrastructure. We codenamed this method, the Network-Attached Accelerator deployment method.
- A complete system architecture that allows an x86-64 master node to allocate, program, and launch computational tasks on the Network-Attached FPGA devices. A mechanism through which the master node sends task descriptions to each individual FPGA board was proposed and implemented.
- We also designed and implemented a Firmware monitor program for managing the internal states of the Network-Attached FPGA boards. Every FPGA board runs an instance of this proposed firmware on a lightweight processor system. The firmware implements supervisory functions such as communication with the mater node and with the worker nodes, dynamically reconfiguring the local FPGA devices, and scheduling operations on the FPGA accelerators.
- The proposed Mechanisms through which the master node sends task descriptions to each individual FPGA boards (NAA nodes) were put into test and verified in an experimental setup where ZedBoards [35] were employed to implement our proposed NAA nodes. The experimental results also verified the proper functioning of the firmware monitor program as well.
- We implemented an accelerator for the Multi-layer convolution operation, which is mainly used in ConvNets, on an FPGA board (NAA node) and compared the computational latency to a software implementation on a Core i7, and we concluded that up to $134\times$ speedup factors can be achieved and up to $1017\times$ energy reductions are possible.

1.3.2 A design methodology for mapping ConvNet inference workloads to FPGA accelerators.

Deep Convolutional Neural Networks are a special kind of deep machine learning models that are typically employed in computer vision systems. In the inference stage, the computations are heavily dominated by discrete 2D multi-layer convolutions, which are basically made up of 2D-convolutions [8]. We propose a design methodology for mapping ConvNet inference to low-end FPGA-based hardware accelerators. The goal of this methodology is to fit complex ConvNets on low-end and resource-constrained FPGA platforms. We tested our methodology on two different resource-restricted FPGA platforms: (1) The ZedBoard [35], and (2) the Cyclone V DevKit [36]; and we successfully deployed AlexNet [13], a popular computational complex ConvNet for image classification that requires 700 million multiplications with 61 million parameters for each image. Our implementations showed accurate results consistent with a non-accelerated software implementation. Images from the ILSVRC-2012 dataset were classified with a top-1 accuracy of 73% and a top-5 accuracy of 84% achieving the FPGA benefits of reduced energy consumption at 0.126 Joules/frame and a potential performance of up to 9 frames/sec as compared to 2.84 frames/sec for a Core i7 processor core (3.16× speedup). Our work introduced several new methods to address and several ConvNet-related design challenges. Those include:

- A novel stride-aware graph-based method targeted at CNNs, and which aimed at achieving efficient signal processing with reduced resource utilization.
- A method to address the challenge of determining the minimal precision arithmetic needed while preserving high accuracy. For this challenge, we

used variable-width dynamic fixed-point representations combined with a layer-by-layer design-space pruning heuristic across the different layers of the deep CNN model.

- A method aimed at achieving a modular design that can support different types of CNN layers while ensuring low resource utilization. we made sure to design small modules that can be interconnected to build an entire accelerator design.
- A method to address the ease of design portability between two different FPGA vendor platforms, namely Intel/Altera and Xilinx

1.4 Thesis Outline

This thesis is organized as follows:

In chapter 2, we cover the essential background needed to understand the basic data center design principles, factors and models. We also provide an adequate introduction to Big data architectures and data management frameworks: (1) Apache Hadoop and (2) Spark.

Chapter 3 provides a detailed explanation of the proposed Network Attached Accelerator (NAA) system architecture. We, first, describe the main internal components of the NAA node. We, then, introduce the NAA compute model, which is used to establish large and complex compute pipelines that can spread across different NAA nodes. Finally, we describe the architecture of the firmware program which runs on the lightweight processor component of the NAA node.

In Chapter 4, we accelerate the Multilayer convolution operation, which is one of the kernel operators employed in Convolutional Neural Networks (ConvNets).

In this chapter, we built an experimental setup to measure the achievable speedup ratio over a software implementation of the Multilayer convolution on a Core i7 processor core. We also developed a latency model for the Multilayer convolution operation and gave insights on the achievable speedup and energy reduction ratios.

Chapter 5 covers an efficient FPGA-based hardware template architecture for Deep Convolutional Neural Network inference is developed. We, first, propose a design methodology for mapping any Convolutional Neural Network architecture to an FPGA hardware accelerator. We employ Graph-based and stride aware signal processing techniques to design 2D convolution circuits with stride configurations. We tested our design methodology by implementing an accelerator for AlexNet on two different resources-restricted FPGA platforms:(1) ZedBoard , and (2) Cyclone V Devkit.

Chapter 2

Background for Data Centers and Big Data Architectures

Enterprise data center technologies have undergone enormous improvements in terms of raw computational power, storage capacity, and reliability. However, current trends in big data, e-commerce, mobile devices, Internet of Things, and massive data mining are pushing the performance envelope beyond the capacity of traditional data center technologies [1, 5, 39]. In this work, we propose to use FPGA-based acceleration technologies to scale up the performance of data center facilities. Augmenting the data center facility with FPGA-based computing platforms requires basic knowledge of the data center environment. In this chapter, we provide a brief introduction to data center architectures, design factors, deployment models, and scalability issues. We also dwell on the stages of big data architectures and adequately introduce the two most important Big data management frameworks: (1) Hadoop [15] and (2) Spark [18].

2.1 Data Center Architectures

A data center is a facility that consolidates an enterprise's IT processes and equipment by housing all the essential components of the enterprise's computing

center [40]. These components can be categorized into three main types: (1) Computational, (2) storage, and (3) networking components. Furthermore, the data center also incorporates efficient power distribution planes, redundant and backup power supplies, cooling units, and numerous security and facility protection devices. The proper planning and designing of data centers is governed by four important factors: (1) Resiliency, (2) Performance, (3) Scalability, and (4) Flexibility [40]. Consequently, many standardization bodies [41, 42] and enterprise networking hardware manufacturers devised multiple design guidelines to properly address the aforementioned factors.

2.1.1 Data Center Design Factors

With the upsurge of digital media and Internet-based services, uninterrupted IT operations became a decisive aspect of most modern successful organizations and enterprises. Organizations that heavily rely on their data center infrastructures to run their business operations are mostly concerned with data center resiliency. Hence providing a reliable data center infrastructure is one of the key factors that are always considered when designing a data center architecture. A resilient data center can minimize chances of service interruption or downtime, potentially saving millions of dollars. A study in 2016 [43] suggests that a data center outage costs an enterprise up to 8000 dollars per minute of outage. Another important factor that plays a major role in data center design is performance. Depending on the application, performance is measured in terms of throughput or latency. Throughput is the number of transactions or computational tasks completed per second. Certain applications require a certain level of responsiveness; in this case, computational latency is the metric of choice when measuring performance..

During the last decade, many applications have increased their computational

demand for the current data center infrastructures; moreover, novel applications are regularly emerging every year. This continuous growth in demand for computational power, regularly forces organizations and enterprises to scaleup the computational performance of their data centers. Consequently, Scalability along with Flexibility need to be cautiously considered when designing a data center infrastructure. A modular and flexible data center architecture allows new applications and services to be quickly deployed in the existing infrastructure resulting in significant competitive advantages for the enterprise [40].

2.1.2 Data Center Network Infrastructure

Many efforts were made at improving the aforesaid aspects i.e., Resilience, Performance, Scalability and Flexibility. The Telecommunications Infrastructure Standard for Data Center ANSI/TIA-942 [41] was the first standard to address the data center networking infrastructure; it provided a flexible structured cabling system and established an official tiering standard for defining the quality of data centers. The ANSI/BICSI 002 standard [42], which describes Data Center design and implementation best practices, specifies multiple design classes of data centers depending on availability requirements and defines a selection methodology that a customer can employ to select a data center design infrastructure class based on the operational availability requirements. Cisco, a world leader manufacturer and vendor of networking and IT equipment, provides a proven layered approach for planning and designing the data center networking component for the enterprise environment. The Cisco layered approach to designing data centers divides the data center network infrastructure into three layers: (1) the core layer, (2) the aggregation layer, and (3) the access layer. The layered architecture of the data center is depicted in figure 2.1.

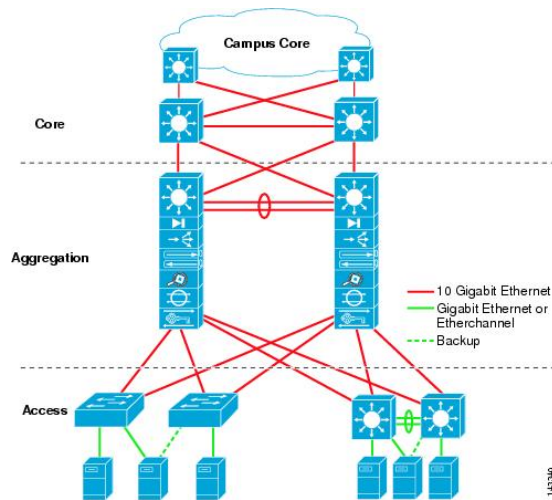


Figure 2.1: Cisco’s Basic Layered Design [40]

The Access layer provides the physical point of connectivity for end-stations, compute nodes, and server resources. In an enterprise data center environment, the server component usually consists of 1RU rack servers, blade servers, cluster servers, or mainframes [40]. The access layer networking infrastructure usually consists of modular Layer 2 and sometimes layer 3 switches. These switches employ a port link aggregation technology called EtherChannel [44] to aggregate server traffic onto 10 Gigabit EtherChannel uplinks to the aggregation layer. Moreover, a common practice in modern data centers is to implement access layer switches paired in groups of two in order to support some level of redundancy in the network infrastructure.

The Aggregation layer is responsible for aggregating thousands of sessions leaving and entering access layer switches to and from the core layer [40], consequently it must be able to provide a high-speed switching fabric. It defines a boundary between layer 3 switching in the core layer and layer 2 VLAN switching in the access layer. A typical data center can be comprised of multiple aggregation modules; as with the access layer switches, each aggregation module consists

of a pair of switches that work jointly to provide redundancy in the networking infrastructure. Multiple aggregation layer services such as firewall, content switching, intrusion detection and server load balancing can all be managed and implemented in an aggregation module by allowing integrated service modules to be deployed. Furthermore, aggregation layer switches are also expected to handle network related overhead processing such as spanning tree and default gateway redundancy protocols.

The core layer provides high-speed layer 3 fast forwarding services and switching between the multiple aggregation modules which together form the data center network infrastructure, the campus core which is the enterprise's regular network, and the rest of the internet. 10 Gigabit Ethernet interfaces are used in the core layer in order to support very high levels of throughput and reduce latency. The data center core layer is an optional layer meaning that it may not be required for small data centers that are not expected to scale in the near future. However, it is recommended to employ a core layer for data center facilities that consist of multiple aggregation modules for reasons related to scalability and performance. To summarize, the practice of dividing the data center network infrastructure into three layers helps in improving flexibility and scalability by allowing the enterprise owner to scale out their data center facilities by flexibly adding more aggregation module. Security may be improved by installing aggregation layer service modules such as firewall and intrusion detection modules.

2.1.3 Data Center design models

Cisco [40] also recommends two different data center design models: (1) The Multi-tier model, and (2) the Server Cluster model. Each design model is suited for different modern use cases. The multi-tier model, which is the most common

model in the enterprise nowadays, is typically used to implement web service applications. Web service applications are normally implemented according to the three-tier client-server model [45] in which the user interface (web service), the application logic (application), and the data storage/retrieval (database) functions are separated into three independent tiers. Consequently, the multi-tier data center model primarily consists of three tiers of servers: web, application, and database servers. In this environment a client-server application is implemented into separate processes that run on different server tiers, and that use the TCP/IP protocol stack to communicate over the network. This separation of servers into different tiers provides some level of resiliency and security to the data center. Resiliency is achieved through redundancy and load balancing between servers. And security is achieved by installing firewalls between the different server tiers.

In the server cluster model, the data center is composed of multiple servers that are linked together through a high-speed network interconnect, and that can act collectively to perform a single complex computational workload. From the user's perspective, the server cluster appears as a single highly-available and fault-tolerant machine with a much higher computational power and storage capacity than what a single server can provide. Although high performance server clusters are usually associated with scientific and military research, current large-scale enterprises and businesses are becoming more aligned with using server clusters in their data centers to implement data- and compute-intensive applications. Examples of demanding applications that are currently being deployed in server cluster data centers are parallel ray tracing algorithms [46] used in the film industry, financial analytics [39] used by financial trading houses, design modeling used in many industries such as the automotive and aerodynamic industries, and finally parallel lookup and page ranking algorithms [28] used in modern search

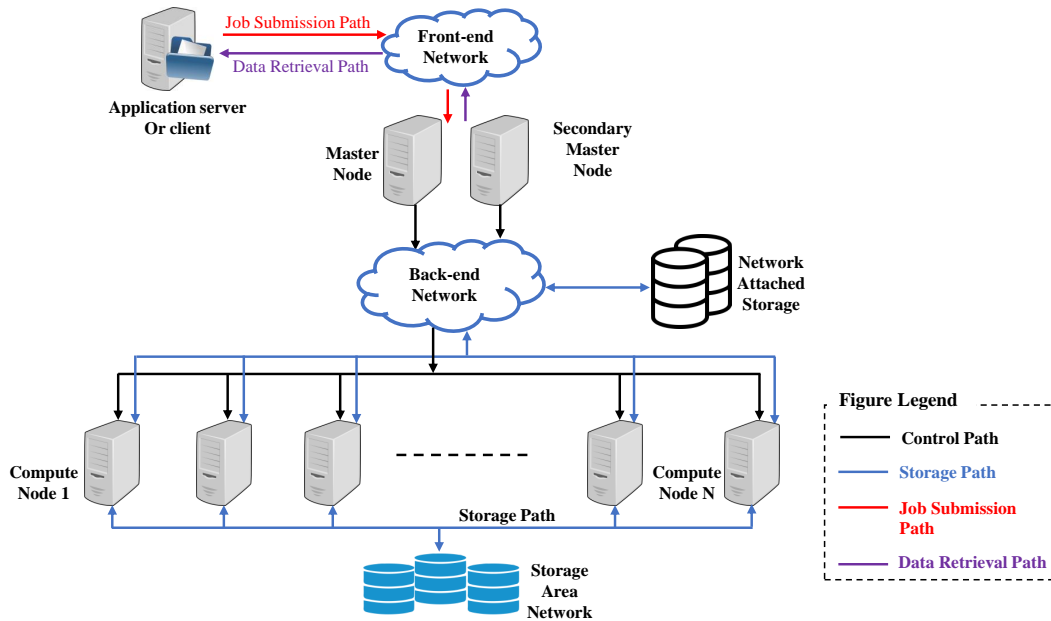


Figure 2.2: A logic view of a server cluster

engines such as Google and Bing. Finally, a new trend that started to unfold during the last couple of years and is accelerating rapidly is the implementation of parallel data mining and machine learning algorithms in server cluster environments [47–49].

Figure 2.2 illustrates a logical view of a server cluster along with its different sub-components. Typically, some of the servers are designated as master nodes and are responsible for scheduling and coordinating computational jobs on the other servers. The remaining servers are commonly referred to as compute nodes, as shown in figure 2. The mass storage sub-component of the server cluster consists of the servers directly-attached hard drive, along with Network Attached Storage (NAS) servers and one or more Storage Area Networks (NAS). A high-speed back-end network provides low latency and high bandwidth connectivity between the cluster’s subcomponents i.e., master nodes, compute nodes, and

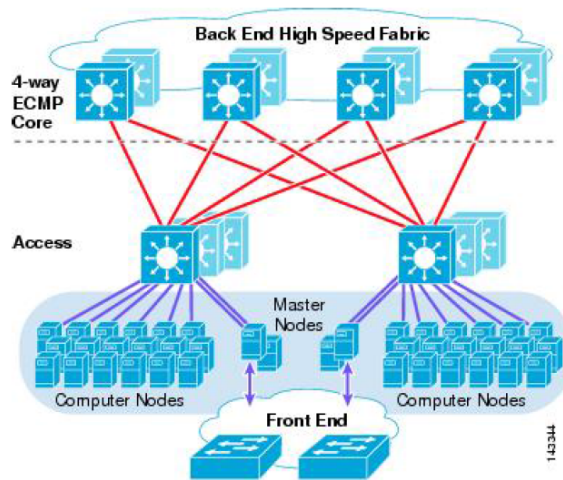


Figure 2.3: A physical view of a server cluster [40]

storage components. The storage path from the compute nodes to the Storage Area Network can be implemented into a separate network such as a Fiber channel network [50], but can also be flowing through the high-speed back-end network. Finally, a front-end network interface is used to provide access to the cluster; application servers or remote clients may submit jobs to the master node and/or retrieve the results of previously submitted jobs through the front-end interface. Figure 2.3, shows a physical view of a server cluster designed according to the Cisco layered approach, but without an aggregation layer.

A clustering middle-ware software running on the master nodes orchestrates all the activities of the compute nodes and allows users to view the cluster as a single computing node by providing the necessary tools and services needed to target the compute nodes. Examples of current cluster middle-ware software tools are Apache Hadoop [15], Spark [citeref:sparkwebsite](#), [ref:sparkpaper](#), and Google Cloud DataFlow [51]. The Server clusters design model is typically associated with High-performance computing, as most of the server cluster use cases relate to massive data and compute intensive applications. Although there are other

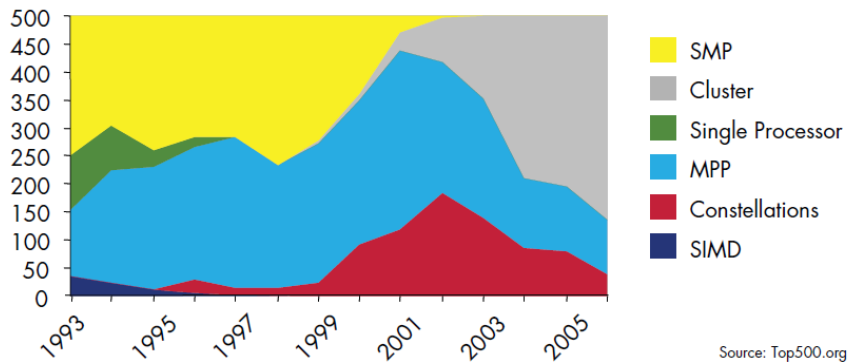


Figure 2.4: HPC Architectures Shift Toward Cluster Computing [39]

high-performance computing architectures, the server cluster model is currently dominating the HPC market due to its cost-effectiveness. Figure 2.4 depicts how the HPC market shifted towards using cluster computing during the last decade.

2.1.4 Data Center Scalability and Power Efficiency issues

As the demand for application performance increases, the computational capacity of the data center facility must scale to meet the demand. There are two different approaches to scaling the computational capacity of a data center facility: (1) horizontal scaling, and (2) vertical scaling. The horizontal scaling approach is colloquially referred to as the “scale out” approach; it usually consists of deploying additional server nodes into the data center facility to match the increase in demand [52]. The other approach, which is the vertical scaling or the “scale up” approach, consists of replacing old and aging server machines with more powerful ones [52]. Historically, improvements in servers’ computational capacity were, mainly, due to enhancements made to the microprocessor’s micro-architecture. During the last decade, however, CPU manufacturers started employing multiple cores on the same chip. [53] Nowadays, improving server performance takes place, mainly, by adding more CPU cores. Subsequently, both approaches, the “scale

up” and the “scale out” methods, involve adding more CPU-bound compute resources.

As mentioned earlier in section 2.1.1, resiliency and performance are among the most important design factors in modern enterprise-grade data center designs. To ensure resiliency and high-availability in their data processing services, enterprises resort to adding redundant compute nodes that act as backup servers; when a server node fails, a backup node may take over. Moreover, the maximum compute capacity of a data center facility must always match the demand for application performance during peak hours. Consequently, data center owners typically plan to “scale up” or “scale out” their data center facilities to handle peak annual demand for server performance [23]; during the rest of the year, a large portion of the deployed server resources remain mostly unused. According to a study conducted by the National Resources Defense Council (NRDC), server utilization remained at around 12 to 18 percent between 2006 and 2012 [23]. Underutilization of CPU-bound compute resources in data centers is one of the most direct origins of data center inefficiencies.

One way of dealing with CPU-bound server inefficiencies is to resort to application domain specific computing platforms such as: General Purpose Graphical Processing Units (GP-GPU), Digital Signal Processors (DSPs), or Field Programmable Gate Arrays (FPGAs). These are computing platforms that are customized for certain kinds of applications and are designed to excel in terms of performance and power efficiency at the cost of a relative loss in flexibility. For instance, GPUs can leverage their highly-parallel architecture to efficiently target parallel algorithms. GPUs, however, may not be used to efficiently implement general purpose workloads, because these workloads might involve computations with a serial and unparallelizable nature; moreover, CPUs might have higher

clock speeds than GPUs. Consequently, GPUs are typically deployed along with CPU-bound servers by tightly coupling a GPU daughtercard to each server machine via a high-speed bus such as PCIe.

During the last decade, FPGA-based application-domain specific computing platforms started to gain more popularity, especially in the field of High-Performance-Computing. FPGA devices are reconfigurable computing chips that generally consist of two kinds of resources: (1) logic, and (2) interconnect resources [24]. Logic resources are comprised of Lookup tables (LUTs) and D-flip-flop elements, organized into logic or functional blocks (LB). Interconnect resources allow multiple logic blocks (LB) to be tiled and connected together to compose a large sequential logic circuit. Both the logic and the interconnect resources are programmable, in the sense that, LUTs and the interconnect configuration can be both re-configured multiple times in the field. As FPGAs started to gain more and more popularity, FPGA manufacturers started adding more features to their FPGA chips, such as embedded hardware multipliers (DSP units), Block Random Access Memory units (BRAM units), on-chip hard memory controllers, and in some category of FPGA chips, dual-core embedded processor systems. Many studies suggest that employing FPGA-based accelerator units in a data center facility has huge positive implications on performance and power efficiency [28, 38, 54].

To summarize, Application-domain-specific computing platforms such as GPUs and FPGAs provide an attractive solution to the problem of scaling the computational power of a data center facility. In this work, we address the use of FPGAs in the data center. In chapter 3, we will develop a method for deploying FPGA-based accelerators in a server cluster data center environment. In chapter 5, we develop an efficient methodology for mapping Deep Convolutional Neural

Networks to FPGA-based hardware accelerators.

2.2 Big Data Architectures

A Big data system architecture is a combination of different complementary sub-systems that together form a wide-ranging value-chain that can be employed by a corporation to derive valuable insights from the data it collects [55]. According to Han et al. [56], a typical big data value-chain consists of four stages: (1) Data Generation, (2) Acquisition, (3) Storage, and (4) Analysis.

2.2.1 Stages of Big Data

The data generation stage is concerned with identifying potential data sources that can be captured and analyzed by the enterprise. Due to technological advancements in digital sensors and information technology, almost all modern business practices rely on amassing colossal amounts of data pertaining to business-to-business, and business-to-customer transactions. Examples of such practices include web-based service applications such as common Enterprise Resource Planning (ERP), and Customer Relationship Management (CRM) solutions that are used by an enterprise to effectively manage and track its business resources. An IDC report [22] estimates that by 2020, internet business transactions will reach a staggering 450 Billion transactions per day. As the amount of data available for businesses expands, the percentage of data that can be curated and analyzed is steadily declining [2]. This growth in business data volume requires more effective big-data-gearred methods to help gaining useful insights in an acceptable time frame. Other sources of data that are currently challenging traditional data processing schemes are networking data and scientific datasets [56]. Networking

data includes, but are not limited to, mobile phones, social media platforms, websites, and IoT devices. At the time of this writing, more than 50% of the world population were reported to have access to the internet [57] and around 30% of them to own smart phone devices [58]. The sheer amount of data generated by social media and mobile phone users exceeds 500 Gigabytes per minute [58].

Han et al. [56] define the Data Acquisition stage as the phase in which all generated datasets captured by an enterprise are aggregated in a digital form that is suitable for storage and analysis. Data acquisition is an umbrella term for three main sub-phases: data collection, transmission, and pre-processing. The data collection method must be carefully considered by the enterprise, mainly because it depends on the physical characteristics of the data source [56]. According to Han et al. [56] data collection methods can be summarized into three methods: (1) sensors, (2) log files, and (3) web crawlers. The collected data is transferred from its place of origin to a data center facility for storage and further processing through high-capacity transmission links. The sheer size of the transferred data and the rate at which it is produced put stringent requirements on the internet backbone through which the data is transmitted as well as on the internal networking infrastructures of data centers. Recently many advancements have been made to address the limitations of traditional data center networking infrastructures. Some of these improvements target the communication physical layer technology used in the data center, such as employing fiber optic cables with wave-division multiplexing, and all-optical switching [59], while other improvements target the transmission control protocol [60, 61].

After collection, the data should be organized and stored in a convenient format that simplifies retrieval and analysis. Storing large trails of data in a data center facility raises two important concerns: the first is locating data within the

data center in a reliable and redundant fashion to insure high-availability and avoid loss-of-data scenarios, and the second is providing a scalable high-speed access interface to the stored data in order not to hinder the overall computational performance of the data center. Accordingly, in a big-data-gear data center, the architecture of the data storage infrastructure and the data management frameworks used are of paramount importance. There are three main storage infrastructure technologies used in current data centers: Direct-Attached Storage (DAS), Storage Area Networks (SAN), and Network Attached Storage servers (NAS) [56]. A typical data center uses a combination of all or a subset of the three storage methods. DAS attempts to extend the storage capacity of single servers by attaching more storage devices to individual servers, while SAN provides a high-speed low-latency interconnect between multiple storage devices and servers. In typical data centers, SAN provides an attractive alternative to DAS, since it offers more flexibility and availability as well as affordable scalability. Lately, however, many successful bigdata platforms [62, 63] employed a distributed storage space strategy; These systems are composed of networked servers with directly-attached storage devices (DAS) and an intelligent data management framework. The data management framework provides clients with an appropriate Application Programming Interface (API) to manage storage, and a parallel programming model to launch parallel computations on the servers. These systems rely on moving the processing function closer to storage devices alleviating the overhead of moving large amounts of data across the network. Regardless of the Storage method used, a scalable, reliable, and an intelligent data management framework is an essential element of most modern big-data-gear data centers.

Finally, the last stage of a Big data value chain consists of analyzing the stored

data. The purpose of this analysis is to extract useful insights from the data and to help a business evaluate its current situation and make the right decisions. Big data analytics rely on computational techniques inherited from multiple disciplines such as statistics and computer science [1]. The most prominently used methods to analyze unstructured Big data are the ones specifically based on machine learning. Lately, Deep Learning methods gained a lot of popularity, as they proved to be very useful and valuable tools for Big data analytics [7]. Deep learning is based on a sub-area of machine learning called representation learning, in which a hierarchical learning process is employed to automatically extract high-level complex representations of the input raw data.

2.2.2 Big Data management frameworks

Apache Hadoop

Apache Hadoop is a highly scalable big data management framework for server cluster data centers based on the master-slave architecture, it allows distributed processing of very large-scale datasets using a simple programming model [2, 15]. In the following we provide a brief overview of the Hadoop project, its common components, its applications and what it can and cannot provide.

Hadoop is divided into four pieces: (1) the Hadoop Distributed File System (HDFS), (2) the Hadoop MapReduce programming model, (3) Hadoop Common, and (4) Hadoop Yarn. Hadoop Distributed File System (HDFS) is a Big data storage facility and framework. In HDFS, data files are broken into data blocks distributed across thousands of interconnected servers. HDFS can make use of commonly available and inexpensive commodity servers, with an inexpensive storage system such as one or two hard drives per server, arranged in a very

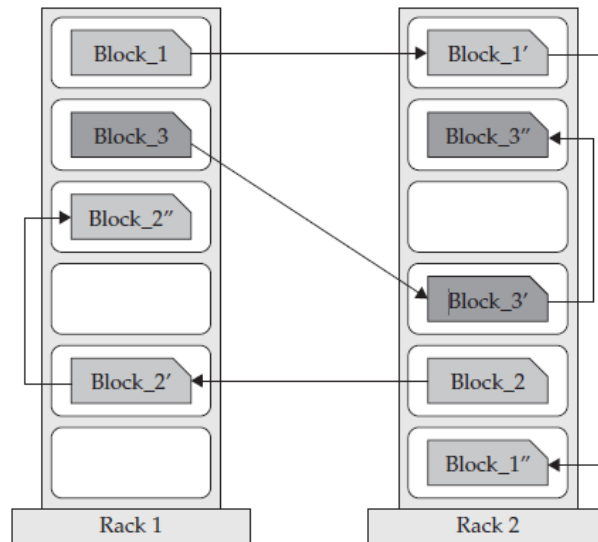


Figure 2.5: HDFS distributing data blocks to rack servers [2]

large network, to create a scalable, robust, fault tolerant and highly-available high-performance computing platform. The high-availability is guaranteed by replicating each data block on another server across the Hadoop cluster [2]. If a node fails, the data blocks residing on this node can still be recovered from the replicas available on other machines. Figure 2.5 shows a typical example of how HDFS duplicates and distributes the data blocks into multiple servers. Note that HDFS may also be rack-aware meaning that it makes it best effort not to place duplicates of the same data block in the same rack of servers. Another benefit of this redundancy is manifested in the ability of the Hadoop cluster to also break a certain computation into multiple tasks and execute those tasks in a distributed, parallel and redundant fashion improving fault tolerance.

In a Hadoop cluster, a single master node manages the data placement logic and keeps all the files' meta-data; The master node is typically referred to as Name Node. The remaining nodes are called Data Nodes and are used to store the aforesaid data blocks in a distributed fashion. Typically, the Name Node stores

meta-data information in main memory with the aim of having a quick response time. A single Name Node is sufficient to manage a huge cluster; however, this creates a single point of failure. To combat failure, the newest versions of Hadoop include the capability to define a Backup Node, which is a standby machine that periodically checks whether the primary Name Node is still alive; if the primary Name Node fails, the Backup Node may take over.

A process, that runs on each Data Node, is responsible for creating storage blocks and destroying them based on instructions received from the Name Node. The Name Node provides an abstract interface and a file system namespace, through which clients can execute filesystem operations such as opening and closing files or creating directories. In other words, if a user desires to upload a certain file to HDFS, the Name Node handles the low-level details such as breaking the file into blocks and determining the mapping of these blocks to Data Nodes. A client who wishes to read from or write to a distributed file in HDFS, must first retrieve the locations of each block from the Name Node, before communicating with the corresponding Data Nodes; the proper Data Nodes can, then, receive read or write requests directly from the HDFS client.

The Apache Hadoop MapReduce framework is a Java-based software framework that provides a programming model and a run-time environment for processing large amounts of data distributed across multiple servers, in a parallel, scalable, and fault tolerant manner. Hadoop MapReduce was inspired by an older implementation of the MapReduce programming model called Google MapReduce [14]. Hadoop MapReduce, however, fixed a lot of the limitations Google MapReduce had. The Hadoop MapReduce framework is built on top of HDFS and uses it as a distributed storage service. The latest version of Hadoop MapReduce uses the Apache Yarn architectural center [64], which supports the notion

of software containerization. Containerization ensures the separation between application logic development and the deployment of applications [65].

In Hadoop MapReduce, a master node, called Resource manager, is responsible for managing the computing resources of the HDFS cluster, and for scheduling/monitoring the execution of MapReduce Jobs. To launch a MapReduce Job on an HDFS server cluster, a client submits a job to the Resource Manager. A daemon process running on the Resource Manager receives the requested MapReduce job and tries to allocate a resource container for executing a job-specific master daemon, called Application-Master (AM). The per-application AM is responsible for negotiating resource containers from the Resource Manager, and for tracking the status of the MapReduce Job. After creating the AM, the Resource Manager tries to contact the Name Node server to locate the data blocks required by the submitted job. Upon locating the data blocks, the Resource Manager breaks down the job into multiple tasks and schedules those tasks on the Data Nodes where the data is located.

A daemon process, called Node Manager, runs continuously on every Data Node. The Node manager is responsible for, creating software containers for MapReduce tasks, monitoring the computing resources of the Data Nodes, and reporting status updates to the Resource Manager. The Scheduler component of the Resource Manager employs the Apache Hadoop Yarn [64]. Yarn supports the concept of Resource Reservation, in which an allocated task can be allocated a set of reserved resources over a certain period of time [65]. Consequently, the scheduling of tasks on the Data Nodes consists of allocating resource containers for the appointed tasks. Figure 2.6 depicts two client processes submitting two different MapReduce Jobs to a Hadoop Resource Manager.

In the MapReduce programming model [14], all data processing Jobs are

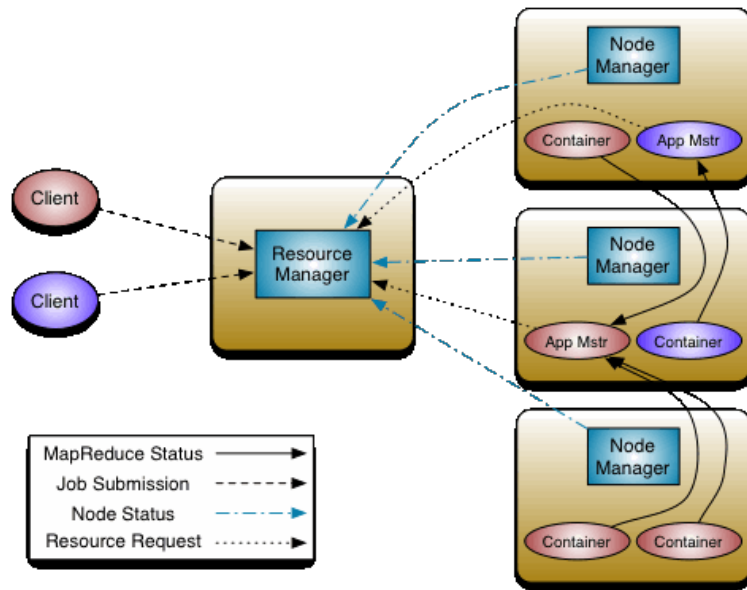


Figure 2.6: two client processes submitting two different MapReduce Jobs to a Hadoop Resource Manager [15]

factored into two stages of computation: (1) a map stage, and (2) a reduce stage. In the map stage, multiple map tasks operate concurrently on distributed Hadoop data blocks, producing intermediate results. These intermediate results are then sorted by the MapReduce framework and transferred to the second stage, namely the reduce stage, in which one or more reduce tasks reduces the intermediate results into the final desired result. The MapReduce framework takes advantage of the data locality principle by scheduling Map and Reduce tasks on the servers where the input data block already exists (Data Nodes), greatly improving performance. In fact, the practice of moving data processing functions to the place where the data block is stored is vastly common among all bigdata frameworks, since the sheer size of the data makes transferring it through the network very undesirable.

In the MapReduce framework the input dataset to a MapReduce Job is always viewed as consisting of Key/Value pairs i.e., a list of pairs in the format (key1,

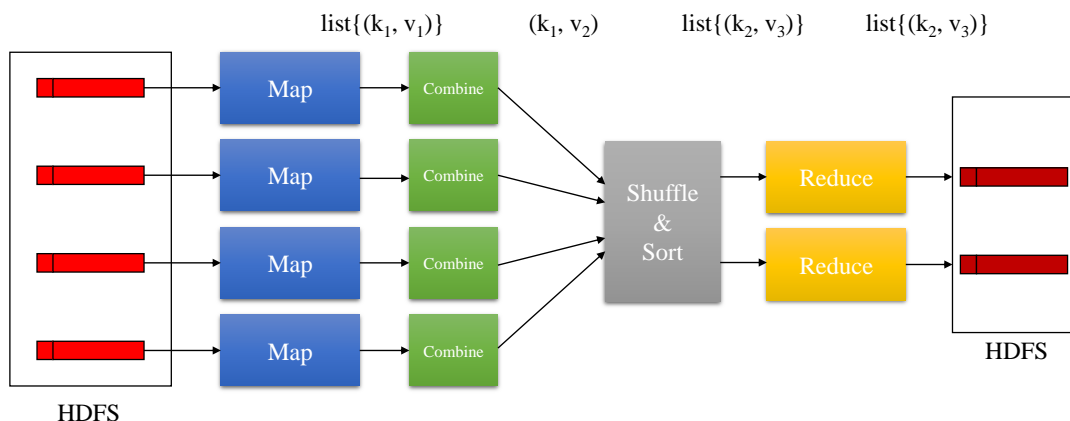


Figure 2.7: Hadoop MapReduce DataFlow

value1) [14]. During the Map stage, multiple map tasks operate on the input dataset in parallel. Pairs, that are processed by the same Map task, are usually assigned the same key. Each Map task also produces a list of Key/Value pairs. The framework, then collects all the pairs produced by the different Map tasks, sorts them, and groups them by key i.e., it creates one group for each key. In the reduce stage, multiple reduce tasks operate on every group of pairs in parallel. Every reduce task may produce a list of output values in another data domain. A programmer, who wishes to write a MapReduce Job, must define both the map and the reduce tasks. In addition to defining the map and reduce subroutines, Hadoop MapReduce may also allow the user to define an optional combine subroutine that locally operates on the output of each individual Map task, before sending results to the reduce stage. The Direct Acyclic Graph, shown in figure 2.7, describes the MapReduce dataflow process.

The Hadoop Common component contains supplementary libraries to support the proper functioning of the aforementioned Hadoop components and to facilitate the interaction with the Hadoop Distributed File System. It includes a file system shell interface allowing users to interactively upload, create, or manipulate

files in the HDFS. Common may also include several other functionalities such as a Native libraries, Rack-awareness support, and security among many others (refer to [15] and [2] for more information on the Hadoop Common component).

Programming with Hadoop requires mastering both HDFS's API and the Map-reduce programming model. Since HDFS and Hadoop MapReduce were both implemented in Java, programmers may also implement their MapReduce applications by implementing appropriate Java interfaces and abstract-classes. Consequently, programming a MapReduce application may require thorough programming skills. To alleviate this difficulty, the Hadoop community developed several Hadoop-related applications or sub-projects that run on top of Hadoop and are suitable for big Data applications such as Mahout [66], Hive [67], Cassandra [68], and many other Hadoop sub-projects.

Apache Hadoop was successful in achieving two big data goals: Scalability and fault tolerance. In practice, however, some applications may require chaining multiple MapReduce Jobs to accomplish the desired functionality. For instance, iterative computations, such as Machine Learning and optimization algorithms, may repeatedly apply a certain function on the same dataset (referred to as the working dataset) [18]. In this case, the computation may be mapped to a chain of MapReduce jobs. Note that, at the end of every MapReduce cycle, data is stored in HDFS and then reloaded for the next cycle. Accessing HDFS is practically translated into disk operations which are relatively slow and may keep Hadoop from delivering “speed-of-thought” response times [2].

Apache Spark

Google MapReduce [14] was first conceived as a parallel and fault-tolerant computing framework for simple batch processing workloads. Although it gained a

lot of popularity, the simplistic nature of MapReduce makes it ill-suited for other types of workloads such as interactive and iterative computations [18]. The limitations of MapReduce led to a variety of specialized cluster computing systems to be developed such as Dryad [16], Pregel [19], and GraphLab [20]. Apache Spark is a unified general-purpose parallel processing engine that can efficiently implement most of the modern types of workloads [69]. The features present in Spark along with its user-friendly interface significantly contributed to the success of Spark. In the following, we begin with describing the essential components of an Apache Spark cluster architecture. Next, we describe the most important characteristics that contributed to the success of this framework.

In a Spark data center, the master server node is, typically, referred to as the driver node, whereas slave nodes are referred to as worker nodes [29]. In addition, to the driver node, a resource manager node is deployed to keep track of all the available compute and storage resources in the cluster. A Spark application consists of many distributed and independent processes coordinated by a single entity called *SparkContext* [29]. Throughout its life cycle, the *SparkContext* object is declared, initialized, and maintained by a Java program, called the *Driver program*. As the name implies, the *Driver program* typically runs on the driver node.

To run a Spark Job, clients submit their user programs to the driver node. A submitted user program, first, declares and instantiates a *SparkContext* object. The *SparkContext* object asks the *cluster manager process*, which usually runs on the resource manager node, to allocate a number of compute resources for the application. The *cluster manager process*, then, launches a number of *Executor processes* on multiple worker nodes, and assign those processes to the *SparkContext* object that requested the compute resources. *Executor processes*

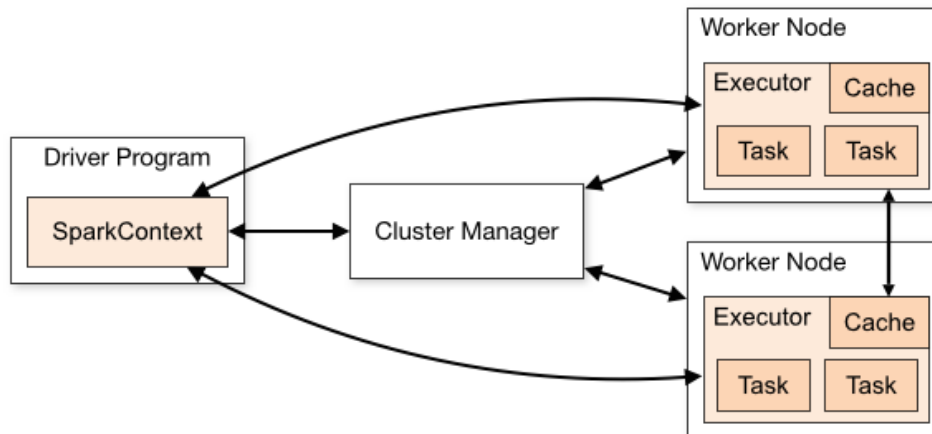


Figure 2.8: Logical view of a Spark cluster [29]

are Java processes whose only purpose is to receive individual tasks or compute functions from the *Driver program* and execute those tasks. The Driver program runs through the application code and depending on the application, tasks are sent to the Executor processes in the form of Java ARchive (JAR) or Python files [29]. Figure 2.8 illustrates a logical depiction of a simple Spark cluster with two worker nodes.

The Spark framework is not restricted to only one type of *cluster managers*. Currently, Spark may be deployed on top of any of the following four cluster management systems: (1) Standalone [29], (2) Apache Mesos [70], (3) Apache Yarn [64], and (4) Kubernetes. The default and the simplest cluster manager for the Spark framework is Standalone. In Standalone, applications submitted to the Spark cluster are executed in first-in-first-out order, and during the execution of each application all the available worker nodes are allocated unless otherwise specified by the user. In Mesos, the cluster manager allows Spark to run alongside other services on the same sever cluster. When Mesos is used as a cluster manager, the responsibility of scheduling tasks is offloaded to the resource manager, because it needs to account for the other services running alongside the

Spark framework. In Yarn, *the Driver program* and the *Executors* all run inside containers. Finally, Kubernetes allows the automatic deployment and scaling of containers. When Kubernetes is used as a *resource manager*, the Spark application may be submitted directly to the Kubernetes cluster, which create a *Driver program* within an abstraction called “Kubernetes pod”, where a “Kubernetes pod” is a group of one or more containers [71]. The *Driver*, then, creates other *Executor* processes within Kubernetes pods. The Scheduling the of execution of both *Driver* and *Executor* codes is resolved by the Kubernetes cluster itself.

As mentioned earlier, *Driver programs* specify the data and control flows of Spark applications. The Spark programming model defines two important abstractions: (1) Resilient Distributed Datasets (RDDs), and (2) Parallel Transformations [18,72]. An RDD is an immutable, in-memory, collection of objects partitioned across many server nodes. A Spark program, typically, consists of a series of user-defined Parallel Transformations, where each Transformation processes one or more source RDD objects and produces one or more different sink RDD objects. RDDs have two interesting properties: (1) they are lazily evaluated, and (2) they are ephemeral [18,72]. Lazy evaluation means that when a *Driver program* encounters a transformation that operates on a source RDD, the resulting sink RDD is not computed immediately. Instead, the framework maintains enough information about the source RDD and the corresponding transformation to compute the resulting RDD, whenever the resulting RDD is needed. Similarly, in the case of a complex chain of transformations that starts with a base source RDD, the framework doesn’t immediately start materializing the RDDs, but instead it maintains enough records that allows worker nodes to compute all the intermediate RDDs along the way until the end of the chain, whenever the sink RDDs are needed. In Spark terminology, these records are stored in a graph-

based structure called Lineage Graph. The Partitions of the lazy RDDs may only be materialized i.e., computed, when a Spark Action on the final sink RDD is invoked. Spark Actions are parallel operations that do not create additional RDDs but are rather used to evaluate a lazy RDD to extract useful information from it.

The ephemeral property means that RDDs get discarded after their first use unless the programmer explicitly changes their persistence property by employing one of the following two Spark actions: (1) **cache()** and (2) **save()** [18]. The **cache** action instructs the framework to evaluate a sink RDD, which will eventually evaluate the chain of all parent RDDs as well, and persistently cache the sink RDD in memory for later use. Similarly, the **save** action evaluates the sink RDD and saves it in stable storage i.e., in mass storage. The choice of using lazy evaluation in RDDs allows the Spark framework to group transformations together contributing to the reduction of both computational complexity and networking traffic between the nodes. In short, when a complex chain of transformations is invoked on a source RDD, the framework composes a Lineage Graph of lazy RDDs. The evaluation of the Lineage Graph, which is basically a Direct Acyclic Graph (DAG) of lazy RDDs, may be triggered by one or more actions [18].

RDDs may be created in one of three ways: (1) by reading its elements from a distributed file system such as HDFS, (2) by parallelizing an existing collection of data in the *Driver process*, or (3) by invoking a parallel transformation on an existing RDD [18]. Examples of commonly used parallel transformations are the **map(*fun*)** and **filter(*fun*)** transformations. The **map(*fun*)** transformation operates on a source RDD and returns another RDD that is formed by passing each element of the source RDD through the function *fun*. The transformation **filter(*fun*)** return an RDD formed by selecting elements from the source RDD

on which the function *fun* returns true. Common examples of actions used in Spark applications are the `reduce(fun)` and `saveAsTextFile(path)` actions. The `reduce(fun)` action aggregates the elements of the source RDD using the user-provided binary function *fun*. The `saveAsTextFile(path)` action saves the source RDD as a text file locally on the Driver node or on a distributed file system such as HDFS. Note that SPARK supports a lot of transformations and actions. The Spark Programming Guide [72] provides a complete treaty on the Spark programming model, including, but not limited to, all available Spark transformations and actions.

In addition to introducing the concepts of RDDs and transformations, Spark introduced two other restricted types of shared variables: (1) *Broadcast variables* and (2) *Accumulators* [72]. In certain parallel applications a large piece of data may be used in multiple transformations and by many worker nodes. Instead of packing the piece of data with every transformation, Spark provides a mechanism through which the piece of data may be broadcasted to all worker nodes only once. To broadcast a piece of data the programmer may create a *broadcast variable* object, and the framework will ensure that the variable is copied to every worker node only once. Some applications require implementing counters or parallel sums, in which case *Accumulators* may come in handy. *Accumulators* in Spark are variables that worker nodes may only add to, through an associative and commutative operator. Associativity and commutativity allow *Accumulators* to be easily implemented in a parallel environment [72].

Chapter 3

Network-Attached Reconfigurable Accelerator System Architecture for the Spark Data Center

The amount of data that is being piled up, stored and analyzed by modern enterprise businesses, along with the data generated by Internet and mobile users on an international scale are growing to an inconceivable degree [1,5]. An IDC report claims that the total volume of digital information created and replicated on the internet will grow to almost 44 trillion gigabytes by 2020 [5]. From the business owner perspective, more collected data means more potentials to gain useful insights and predict future outcomes, yet only a small portion of this data can be analyzed [2]; Moreover, this portion of data is growing smaller everyday as the rate at which the data is proliferating is gradually outpacing the computational capacity of modern data centers.

Historically, the growth in demand for application performance in data center infrastructures was matched with improvements in the microprocessors' computational capacity. For most of its history, the microprocessor's computational and thermal performances remained ahead of demand [39]. Lately, however,

this situation started to change dramatically as most of current businesses and enterprises became highly data-driven. Moreover, according to Intel [73], the pace of advancement in transistor density in monolithic integrated circuits has slowed down since 2012 and this trend will continue thereafter. To respond to this performance gap between the insatiable demand for application performance and the microprocessor sheer computational capacity, business owners and enterprises scale up and out their data center facilities by adding more processor cores to meet the demand. This strategy, however, made the data centers very inefficient in terms of energy and power consumption. According to an NRDC report published in 2012, the main source of inefficiency in current data centers is CPU underutilization. The study concluded that server's utilization remained between 15 and 18

To mitigate the problems of CPU-bound server clusters, the High-Performance Computing (HPC) community started searching for application-domain specific computing technologies that can harness the intrinsic composition and parallelism in their applications [25, 74]. The most famous application-domain specific computing platform was the Graphical Processing Unit (GPU). After years of success in the gaming and research industries, GPUs started to find their way to the HPC and data center markets, due to their parallel structure and their capacity to efficiently target parallel algorithms.

Employing custom-designed Integrated Circuits (ASIC), is another possible alternative to CPU-bound architectures. Although custom-designed ASIC-based solutions always provide the best performance per energy and power figures, employing them in the data center environment is nearly impractical, because data center services evolve so rapidly, and ASIC fabrication processes are relatively slow. Recently, the HPC community realized that they can use reconfigurable

computing platforms such as FPGAs to build custom-designed accelerators for a wide-spectrum of workloads. FPGAs provide a very good balance between hardware acceleration, customizability, and flexibility, through the concept of re-configurability [24]. A lot of research suggest that FPGA-based implementations of HPC workloads may provide many benefits in terms of performance, and power efficiency [28, 38, 54].

Aiming at advancing the deployment of FPGA-based accelerators in the data center environment, we propose an FPGA-based Network-attached Accelerator deployment model for the server cluster data center environment. The proposed architecture is also aimed at supporting currently available big data management and computing middle-ware systems, such as Hadoop MapReduce [75] and Spark [18, 29]. The contributions of this work are as follows:

- A Network-Attached Accelerator (NAA) deployment model for the server cluster data center, in which Accelerator nodes are centered around an FPGA device and are attached directly to the cluster’s network infrastructure. Although a processor system is employed in the accelerator node, the processor system is lightweight and is only meant to implement supervisory functions.
- Due to the network-attached nature of the proposed system architecture, the NAA cluster may be efficiently scaled out by simply adding more NAA compute nodes to the cluster infrastructure.
- We proposed an NAA compute model that allows for tasks to be distributed across multiple NAA compute nodes. This model permits the Spark Driver node to establish a long computational pipeline that may extend across multiple NAA compute nodes.

- A Firmware architecture for managing the low-level aspects of the accelerator nodes. The firmware runs on the lightweight processor system and implements supervisory functions such as communicating with the server nodes, reconfiguring the FPGA device, and scheduling operations on the FPGA-based accelerators.

3.1 FPGA-based Deployment Models

Methods for deploying FPGA-based reconfigurable computing platforms in the data center environment can be divided on the basis of the coupling-level between FPGAs and microprocessors or the attach technology [37]. In general, there are roughly two main categories of deployment methods: (1) Co-processor deployment, and (2) Network-attached accelerator deployment. In the first category, which is the co-processor deployment methods, an FPGA device is attached as a separate module that is tightly-coupled with the processor's system bus. The FPGA typically implements a custom-designed accelerator for a certain computational workload that the client desires to speed up. A key benefit of employing a co-processor deployment method is the ability to take advantage of the high-speed data pathways that the tight-coupling between the FPGA and the microprocessor may provide. Thus, legacy compute intensive applications can leverage the existence of a tightly-coupled FPGA-based accelerator to speed up the execution of a selected list of computational kernels, in which case a simple device driver may be used to abstract the FPGA low-level interface and provide an extensible API for the application to target the accelerator. Typical, implementation examples of co-processor methods employ FPGA boards with PCIe edge connectors [28, 38] attached to server nodes, or motherboard implementations that

deploy Xeon processor chips along with FPGA devices tightly-coupled to the CPU system through Intel's Quick-Path Interconnect [37]. A key disadvantage of these methods, however, is scalability; as inserting a new FPGA device into a server cluster always requires a CPU-based server to host it. Moreover, typical HPC workloads require a significant number of FPGA devices to meet the ever-increasing computational demands of modern applications. Consequently, the number of FPGA devices may not significantly surpass the number of CPU-bound x86-64 nodes. Given that one of the most prominent sources of energy inefficiency in modern data centers is CPU underutilization [23], having a CPU-bound x86-64 node attached to every FPGA could be a great disadvantage to employing the co-processor model.

The second category of deployment methods relies on connecting FPGA-based accelerator modules directly to the Data center's network infrastructure. In this deployment model, an FPGA-based accelerator unit is an independent and self-contained compute node that can provide computational services to other entities in the data center. Consequently, an FPGA-based module should exhibit all the necessary interfacing technologies to allow it to connect to the network infrastructure such as an Ethernet Network Interface Card (NIC). The accelerator module should be a standalone system with enough intelligence to operate on its own without a direct fine-grained intervention from a server node. Consequently, those nodes may include a lightweight embedded-style microprocessor to implement node and job management functions. The microprocessor may also receive a coarse-grain job description from a master server node. This kind of deployment methods can be scaled out flexibly, since the addition of an accelerator module doesn't require a directly-attached x86-64 server node to host it.

3.2 Related Work

Deploying FPGAs in the data center environment has captured the attention of both industrial and academic research groups. In the following, we will shed some light on the latest research endeavors to develop and deploy FPGA-based solutions for the data center. Both the co-processor and the network-attached deployment models were investigated.

3.2.1 Co-processor implementations

Microsoft's Catapult architecture

The Catapult reconfigurable fabric [28] was among the first attempts at developing FPGA-based co-processors to accelerate compute and data intensive data center workloads. It was developed by Microsoft to accelerate document ranking, which is a compute intensive workload used in the Bing search engine. The approach took by Microsoft relies on supplementing every x86-64 server in a server cluster data center with a small FPGA daughter card. The daughter card is interfaced with the server's motherboard via a PCIe edge connector to minimally disrupt the server's architecture. A secondary network connects the FPGAs directly. The purpose of connecting FPGAs directly was to provide a low latency and high bandwidth inter-FPGA network. This allows complex applications, that require more than one FPGA, to be mapped across multiple FPGAs efficiently.

Microsoft was faced with two challenges when deploying FPGAs in the data center: the first challenge was to meet the low latency and high bandwidth requirements of the secondary inter-FPGA network; the second challenge was to marginalize operational expenses when servicing their FPGA-augmented machines. To address those challenges, they selected a two -dimensional 6x8 torus

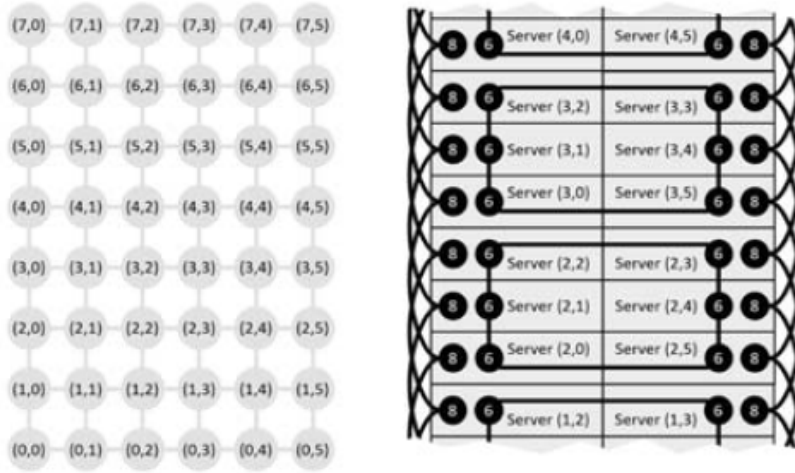


Figure 3.1: Catapult’s 6×8 torus network topology and the physical wiring on a pod of servers [28].

topology for the inter-FPGA network. They also routed all the FPGA daughter card’s high-speed traces to the back of each server chassis and connected them to a Serial Attached SCSI (SAS) port. The SAS port may be plugged to a passive backplane, which held the 6×8 torus cabling. This configuration provided an easy way for servicing FPGA-augmented servers by allowing the maintenance team to pull it out of the backplane without dealing with complex cabling. Figure 3.1 depicts the logical 6×8 torus network topology used in Catapult reconfigurable fabric.

The Catapult employed high-end Altera Stratix V D5 FPGA devices in the daughter cards. These devices have substantial logic resources such as Block RAMs, DSP units and Look up tables. The high-resource density of these devices allowed the designers of the Catapult to logically divide the FPGA fabric into two partitions: shell and role partitions. The shell consists of all reusable components that are common among all applications. The main elements of the shell: (1) DRAM controllers, (2) serial links with a lightweight communication

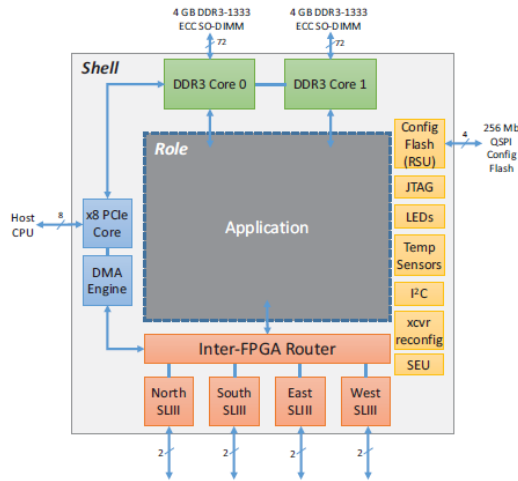


Figure 3.2: Components of the Shell Architecture [28].

protocol for the inter-FPGA network, (3) router logic to manage inbound and outbound traffic from the PCIe connector, (4) reconfiguration logic, (5) the PCIe core. The role consists of the application logic itself. The study found that FPGA acceleration benefited the document ranking by providing lower latency and better scalability. According to the study, a major challenge in the long term is the programmability, and the development of FPGA designs, since it still requires hand-coding RTL and manual tuning. Although, Catapult, can benefit the data center in terms of performance and efficiency, the study doesn't discuss a suitable parallel programming model for the framework, and was limited to page ranking. Figure 3.2 illustrates the division of the FPGA fabric into the shell and role components.

Intel's Xeon+FPGA architecture

The exponential growth of mobile data traffic across the web is alone fueling the data center growth. This emerging trend is urging leading high-performance computing manufacturers such as Intel to search for architectures that incorporate

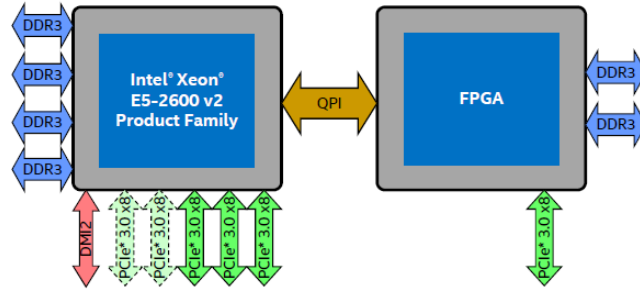


Figure 3.3: Intel’s Xeon+FPGA platform [37].

reconfigurable computing elements in order to improve performance and achieve better energy efficiency figures. In 2015, Intel announced that it has completed the acquisition of Altera, a historically well-known leading FPGA manufacturer. Altera is now part of Intel’s Programmable Solution Group which is working in tandem with Intel’s Data Center Group to deliver the next generation of FPGA-geared data center architectures [76]. Recently, Intel proposed a new platform for data centers that incorporates an Intel Xeon processor tightly-coupled to an Altera Stratix V FPGA through a Quick Path Interconnect (QPI) [37, 77]. The FPGA acts as a co-processor and is coherently attached to the microprocessor system allowing the FPGA fabric to have easy access to the CPU’s virtual memory system. In addition to allowing the FPGA core to access virtual memory, the platform enables efficient implementations of hybrid processing models with fine grained interaction between the processor and the FPGA. Figure 3.3 depicts the architecture of Intel’s Xeon+FPGA platform.

IBM PureData System

IBM delivers a product called IBM PureData System [62] for Analytics Architecture targeted at database, processing and storage. The PureData system is based on the Netezza architecture, which uses FPGAs near the disk I/O to fil-

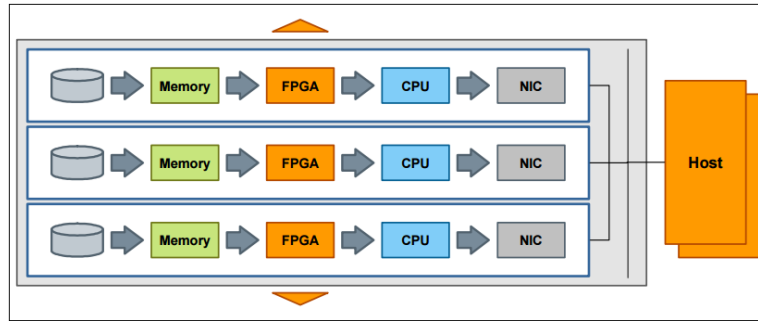


Figure 3.4: A Snippet Blade [62].

ter out superfluous data as it streams off the disk. The filtering frees the rest of the system from processing unwanted data, and hence significantly enhancing system's performance. The PureData system employs a novel architecture, which combines an SMP front-end and a Massively Parallel Processing (MPP) back-end. The SMP front-end, known as SMP host is a high-performance Linux server responsible for compiling SQL queries into executable code segments for processing on the Snippet Blades, which are MPP engines, each with an FPGA for filtering streams of data as mentioned before. Snippet Blades also incorporate a multi-core CPU, high capacity RAM and a network interface card. All the price-performance advantages of the system are due to an embedded engine implemented in the FPGA called FAST. FAST contains a Compress engine which un-compress disk blocks at wire speed, transforming a disk block into 4 to 8 blocks in memory. FAST also contains engines responsible for filtering out the data based on the SQL query.

Xilinx SDAccel Environment

SDAccel is an Development Environment for targeting FPGA-based acceleration from Xilinx [38]. It provides data-center-application developers with the necessary hardware and software design tools to target FPGA-based accelerators in the

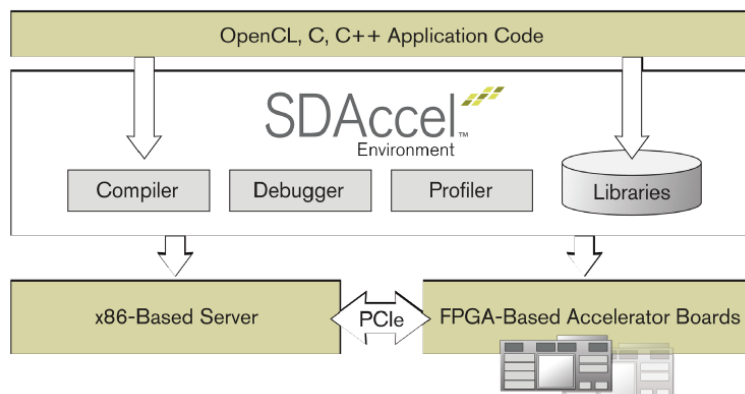


Figure 3.5: The SDAccel Development Environment [38].

server cluster data center environment. The SDAccel development environment is similar to the GPU work environment; it allows the application developer to dynamically and flexibly deploy and decommission accelerators in run-time by using the concept of partial dynamic reconfiguration. The SDAccel compiler targets x86-64 server machines augmented with FPGA-based PCIe cards. The compiler supports any combination of OpenCL, C, or C++ application specification languages and generates optimized hardware architectures for streaming and low-latency computations. The most important contribution Xilinx accomplished with SDAccel is giving software developers access to a familiar workflow that allows them to exploit the raw computational power of FPGAs with little to no prior FPGA experience. In addition to providing an efficient compiler tool, the Integrated Development Environment (IDE), offers the ability to emulate FPGA devices on the x86-64 server machines. Emulation allows developers to easily test and verify their designs for functionality, before deploying them on data-center-ready FPGA-based PCIe cards. Figure 3.5 depicts the layered architecture of the SDAccel development environment.

SDAccel’s compiler was tested with real world benchmark applications such as compression and encryption and was able to produce $3\times$ smaller and $3\times$

faster designs than any other design generated using other High-level Synthesis tool. Moreover, the generated designs were comparable to hand coded Register-Transfer Level (RTL) designs. Although SDAccel provides a GPU-like programming environment, an emulation facility and a powerful compiler, it relies on a co-processor deployment model. We mentioned earlier that, in the co-processor deployment model, it is necessary to have a CPU-based server node to host every FPGA daughter card; Consequently, scaling out the FPGA facility necessarily requires adding more CPU-bound server nodes.

3.2.2 Network-Attached Accelerator implementations

FPGA Accelerated Storage Architecture for Data-Intensive Applications

An FPGA-based Solid-State Drive architecture, called Reconfigurable Active Drive (RAD), was proposed by Li et al. [78]. The RAD combines the processing power of FPGA-based accelerators and the high-bandwidth access of a Solid-State Drive. Similarly, to what was proposed in the IBM PureData architecture, RAD was developed based on the idea of moving data processing elements i.e., FPGAs, closer to the mass storage facility. In order to achieve very high throughput between the SSD storage elements and the FPGA, RAD employs an array of parallel flash chips. An FPGA device is placed between the SSD controller and the flash package array as shown in figure 3.6. The FPGA may be divided into two components: (1) User logic, and (2) Flash-access related logic. The User logic is used implement data-intensive applications such as database processing and data mining. The Flash-access related logic is used implement SSD components such as a flash multiplexer and SSD control logic.

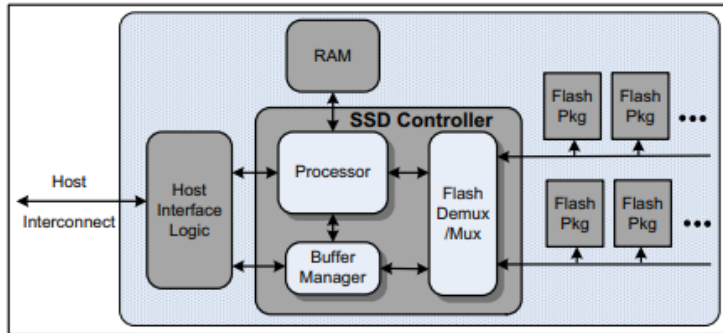


Figure 3.6: The RAD architecture [79].

RASSD: Reconfigurable Active SSD Platform for Data Intensive Applications

Along the same lines, Abbani et al. [54] proposed another FPGA-based Reconfigurable Active SSD architecture, codenamed RASSD, that can be deployed in a data center environment. A RASSD node brings together one FPGA device and an SSD, in a tightly-coupled configuration. The FPGA device is used to implement an accelerator that can process data streams from the SSD component. Figure 3.7 illustrates the architecture of a RASSD node. In the data center environment, RASSD nodes may be organized into clusters of compute nodes; each cluster of RASSD nodes may be managed by one middleware server node (MWS). MWS nodes are responsible for receiving compute requests from client applications, and for orchestrating the execution of workloads on the RASSD nodes [80].

The FPGA device on the RASSD node includes a MicroBlaze soft processor core, peripheral controllers, and a user-defined reconfigurable region. The reconfigurable region (c.f. figure 3.7) is used to place a custom-designed accelerator to accelerate client’s workloads. RASSD supports dynamic partial reconfiguration meaning that the reconfigurable region may be separately reconfigured without

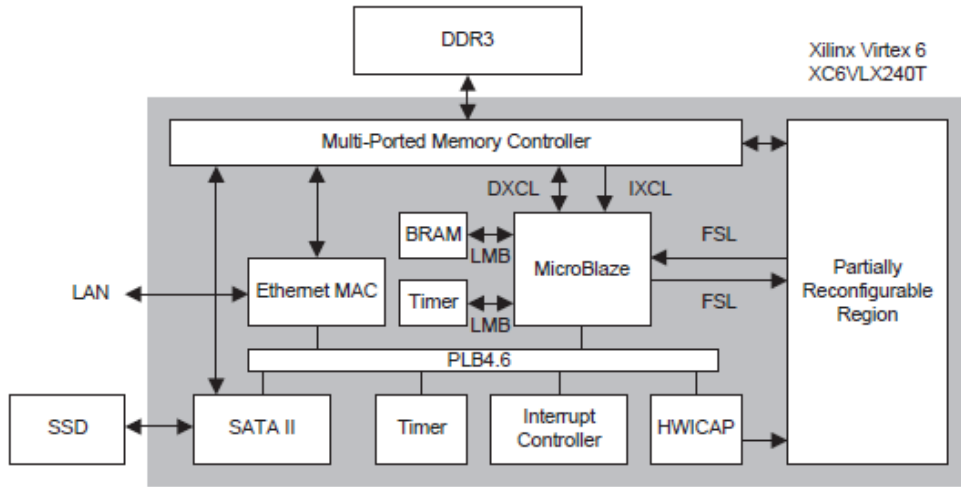


Figure 3.7: RASSD node prototype [54].

affecting the remaining FPGA components. The MicroBlaze soft processor core runs a lightweight Monitor program that performs supervisory function and executes accelerator driver codes called drivelets [54, 81]. A drivelet is a piece of software code sent by a Middleware server to a RASSD nodes to execute a certain workload on the MicroBlaze soft processor. The RASSD was tested using K-means, and it was found that it can run 1.3 to 15.2 times faster than a multi-core CPU processor implementation while consuming 9.4 to 201.9 times less energy.

To deploy the RASSD node in the data center environment, a Hadoop extension to the RASSD platform was proposed in [82]. The extension supports data distribution and execution of mappers and reducers on the RASSD nodes. Moreover, a performance model was proposed to evaluate the impact of the proposed extension. In [83], a multi-tasking and real-time operating system architecture for the RASSD platform was proposed. The RASSD operating system is intended to abstract and hide the low-level details of the RASSD platform from the running applications.

3.3 Proposed Deployment Model

3.3.1 Challenges

1. The computational workloads involved in data centers are typically complex and require deploying many FPGAs to meet the desired computational capacity.
2. The demand for application performance is constantly on the rise; consequently, any successful deployment model should be able to scale out and up easily to meet the demand. Where scaling out refers to the ability to flexibly increase the number FPGAs in a server cluster and scaling up refers to the ability to deploy more power FPGA devices.
3. The deployment model should be able to support a wide range of FPGA devices.
4. The cluster resource manager should be able to dynamically reconfigure any FPGA node remotely. Given that our implementation belongs to the category of Network-attached Accelerators, which usually don't involve a high-end CPU core, it is essential to develop a low-overhead framework that runs on each individual FPGA node. The framework should enable many supervisory functions, among which is communicating with the resource manager across the network, and reconfiguring the FPGA fabric dynamically.
5. The deployment model should be able to easily integrate with most big-data management frameworks such as Apache Hadoop, and Spark. Consequently, the architecture of the FPGA-augmented server cluster should be

analogous to that of Hadoop or Spark.

To address those challenges, we propose a scalable model for deploying FPGA-based compute nodes in the server cluster data center environment. We will, first, describe the system architecture of an FPGA augmented server cluster; we code-named our architecture “Network-Attached accelerator” or NAA for short. We will, then, propose a lightweight, scalable and flexible firmware architecture for managing NAA nodes. The proposed firmware allows the server cluster manager to gracefully add or decommission NAA nodes during the operation, thus enabling scalability. In this model, we specifically targeted the Spark cluster computing system; consequently, we devised a layered software approach for deploying the accelerator in the Spark environment.

3.3.2 Network-Attached Accelerator system architecture

Figure 3.8 shows the functional architecture of a server cluster data center augmented with FPGA-based Network-Attached Accelerators (NAAs). This architecture was inspired by the structure of the Apache Spark cluster computing system [18, 29]. As in a Spark server cluster, the proposed architecture involves a master node, a resource manager, and many slave server nodes. In addition to the slave server nodes, this proposed architecture incorporates many Network-Attached Accelerators (NAAs) and a Bitstream Repository.

An NAA compute node is a Single Board Computer System centered around an FPGA device. The FPGA device can implement a custom-designed accelerator for an arbitrary computation; moreover, the FPGA fabric may also implement other supervisory functions. To run a certain application on the NAA compute nodes, the driver program running on the master node asks the NAA resource

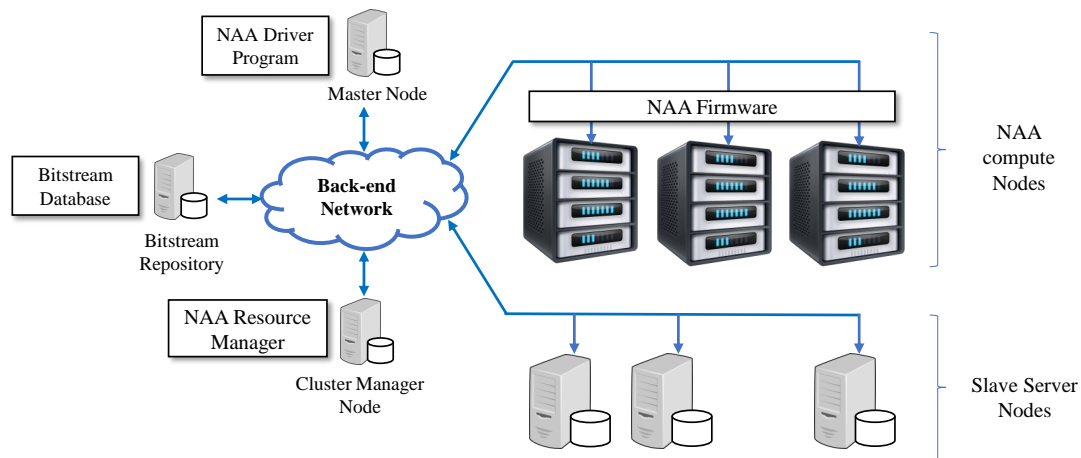


Figure 3.8: NAA system architecture

manager, which is running on the Cluster manager node (c.f figure 3.8), to allocate a few NAA compute nodes for the application. After allocating those NAA nodes, the driver program contacts the Bitstream Repository to fetch suitable bitstreams for those nodes. The driver program then sends the designated bitstreams to the allocated NAA nodes and instructs them to program their FPGAs using the abovementioned bitstreams. Upon receiving the suitable bitstreams and programming their FPGAs, the designated NAA compute nodes notify the driver program that they are ready for operations. The driver program, then, attempts to establish a computational pipeline using the allocated NAA compute nodes. In section 3.3.4, we will describe the concept of the NAA program and how its provide an elementary mechanism for establishing the aforementioned pipeline. Once established, the pipeline can be fed with data from two sources:(1) from the slave server nodes, or (2) from the NAA’s on-board DRAM memory chips.

An instance of a firmware monitor program is installed on every NAA compute node. The firmware monitor program is responsible for managing all aspects related to scheduling computational tasks on the FPGA device, monitoring the status of the compute node, sending synchronization heartbeats to the NAA

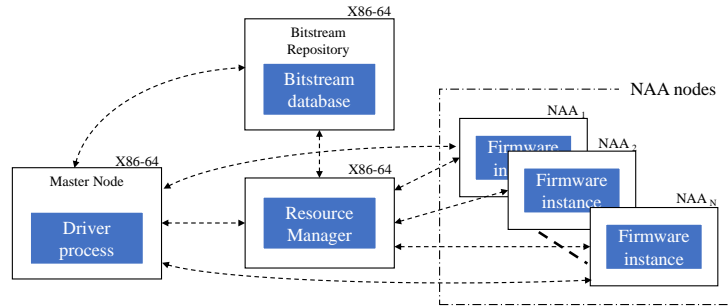


Figure 3.9: NAA system logical view

Resource manager and the NAA driver program, establishing communication channels with the driver program, or any other slave server node. Figure 3.9 depicts a logical view of our FPGA-augmented server cluster along with its main elements.

3.3.3 NAA Node Architecture

An NAA compute node consists of three components: (1) a general-purpose multi-core microprocessor component, (2) a reconfigurable component (FPGA), and (3) a Networking component. The networking component provides a way to integrate NAA compute nodes into the server cluster’s network infrastructure. The FPGA component offers a dynamically reconfigurable fabric that can be used to deploy custom-designed hardware accelerators for arbitrary computations. The general-purpose multi-core microprocessor component implements supervisory functions and is used to run our proposed firmware architecture. We will further dwell on the structure and role of our proposed firmware architecture in section 3.3.5.

Depending on the FPGA device used, NAA compute nodes may range from low-end embedded-style System-on-a-chip (SoC) platforms, to high-end FPGA devices with very large resource counts. NAA compute nodes may also be classi-

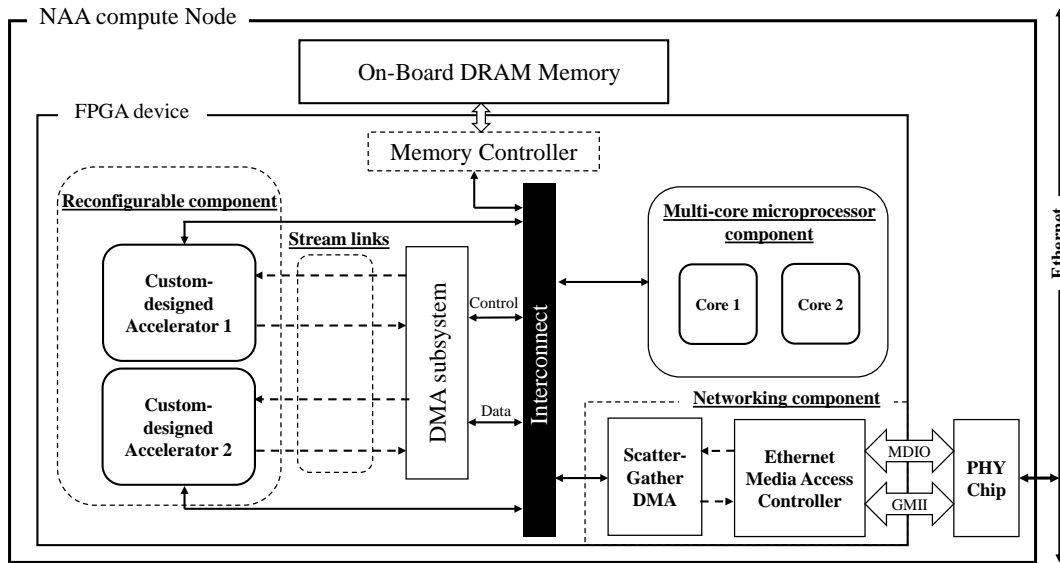


Figure 3.10: NAA node Architecture.

fied based on the multi-core microprocessor component used. At one end of the spectrum, an NAA node may be built around an FPGA device with no embedded microprocessor whatsoever; in this case, a soft-processor core may be deployed in the FPGA fabric to run the supervisory functions. At the other end of the spectrum, an NAA node may include a full-fledged CPU chip tightly coupled to an FPGA device, in which case the CPU may consist of a multi-core x86-64 microprocessor chip, while the FPGA is tightly coupled to the CPU's memory space through a high-speed interconnect. Figure 3.10 illustrates the architecture of a typical NAA compute node and shows how its different components are assembled together to form a complete processing module. An interconnect is used to bring the various components of the NAA compute node together (c.f. figure 3.10). A memory controller is used to interface with the on-board DRAM memory, which may be a hard or a soft controller depending on the FPGA platform used.

As implied in figure 3.10, the Reconfigurable component may include more

than one custom-designed hardware accelerator simultaneously (c.f. figure 3.10) and each accelerator may be accessed through a different set of input/output streaming interfaces. A multi-channel Direct Memory Access (DMA) subsystem is employed to couple the hardware accelerators to the on-board memory, through the interconnect and the memory controller. The DMA subsystem has the capability to read a block of data from on-board memory and send this block as a pipelined stream through a specific stream interface to one of the custom-designed accelerators. Similarly, the DMA subsystem may also read a pipelined stream from a particular stream interface and write it to a pre-designated memory location. The microprocessor component running the previously mentioned firmware monitor program, is responsible for scheduling memory-to-fabric and fabric-to-memory data transfers. In addition to scheduling data transfers, the firmware also runs other supervisory functions such as communicating with the NAA resource manager and with the NAA driver process (c.f. figure 3.8), configuring the reconfigurable fabric component with new bitstreams, and managing the meta-data of the hardware accelerators.

3.3.4 NAA Compute Model

In this work, we employ the streaming dataflow compute model [24] to reason about the inherent composition and parallelism in an accelerated application. Recall that in a streaming dataflow compute model, the computations are modeled as Directed Graphs (DGs), in which case operators or sub-computations are modeled as vertices, and data transfers between operators are modeled as edges; in other words, a computation is modeled as a directed graph, in which the output of one or more operators are linked to the inputs of one or more other operators. We specifically model our accelerators using the Parametrized Dataflow Model

of Computation (PDMoC) [84]. In the parametrized streaming dataflow model, some operators may be dynamically configured by re-assigning a set of parameters. A reconfiguration is said to have occurred whenever the parameters of a parametrized Dataflow Graph are re-assigned; moreover, this reconfiguration may occur only at certain points during the execution of the graph in order to preserve the integrity of the output [84, 85]. Modelling computational tasks as such may have a lot of benefits, as this Model of Computation (MoC) has a very high expressive power and it may also allow data-path and interconnect sharing to take place when implemented. Accordingly, in this study, a custom-designed accelerator is modeled as a transformation T that operates on a set of inputs X and a set of parameters P to produce a set of outputs Y . The output of an accelerator is mathematically expressed as follows:

$$Y = T(X, P) \tag{3.1}$$

Where $X = [x_0, x_1, \dots, x_{n-1}]$, $P = [p_0, p_1, \dots, p_{m-1}]$, and $Y = [y_0, y_1, \dots, y_{q-1}]$. T is a general transformation that represents the accelerated computation. Consequently, every custom-designed accelerator exhibits one or more parameter stream interfaces, one or more input data stream interfaces, and one or more output data stream interfaces. Figure 3.11 shows an archetypal custom-designed accelerator with one parameter interface, one input data interface and one output data interface. The parameter interface is used to configure the accelerator for a specific instance of the computation. This is done by loading an array of parameters through the parameter interface. The input data interface is used to stream the input data into the accelerator, while the output data stream interface is used to capture the results of the computation. In addition to the parameter, input

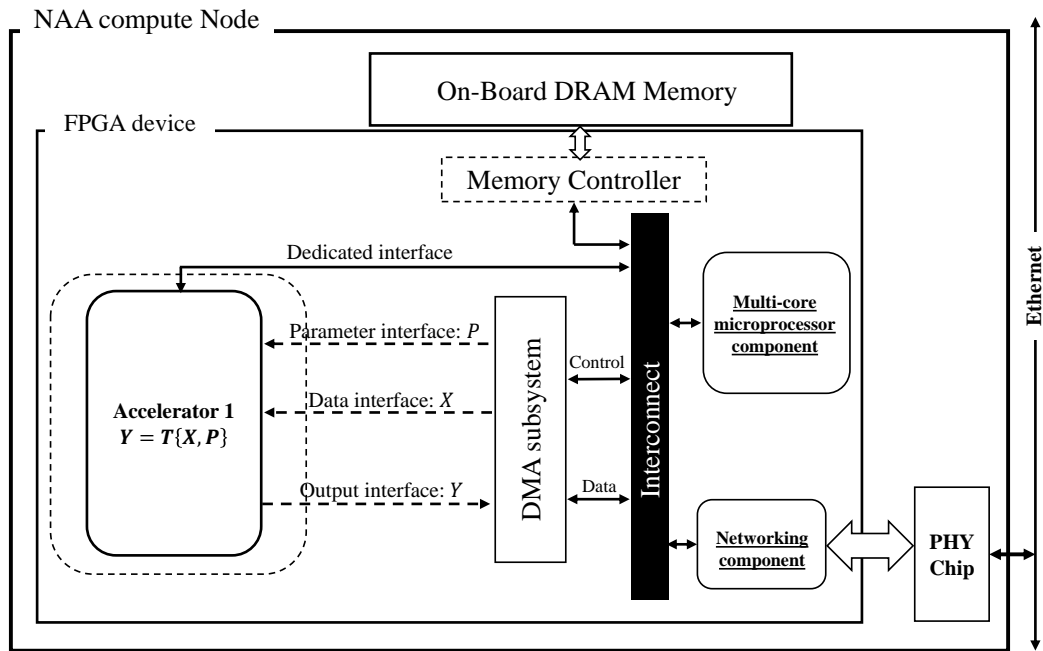


Figure 3.11: Accelerator’s interfaces: parameters, input and output interfaces.

and output interfaces, during a computation, the accelerator may need to access the On-board memory to store intermediate data. Consequently, the accelerator may also communicate with the On-Board DRAM memory through a dedicated interface that bypasses the DMA subsystem (c.f. figure 3.11). The firmware is responsible for allocating dedicated memory buffers for the accelerators to use during their life cycle.

Internally, an accelerator can be in one of two states: *unconfigured*, and *active*. Initially, the accelerator will be in the *unconfigured* state. In the *unconfigured* state, the input data stream interface is blocked; in other words, it is not allowed to receive any input data, since the internal circuits are still not configured to perform the appointed computation. Upon receiving configuration parameters via the parameter interface, the accelerator changes its internal state to *active*. In the *active* state, the parameter interface is blocked, whereas the input data

stream interface is allowed to receive input data, and the internal circuits of the accelerator are allowed to process this stream of input data and produce a resulting stream of data at the output interface. The number of tokens consumed at the input data interface, and the number of tokens produced at the output interface are also parameters received during the *unconfigured* state. When the accelerator concludes the computation, it jumps back to the *unconfigured* state and waits for another set of configuration parameters to be received on the parameter interface.

One example of a PDMoC-based accelerator is a two-dimensional Finite Impulse Response filter with a re-adjustable filter mask size and re-adjustable coefficients. In this case, both the filter mask size and the coefficients of the filter are parameters that can be set using the parameter interface. Another parameter that can be set in this accelerator is the size of the input image that the filter needs to process. In chapter 5, we dwell more on designing convolution filters for deep convolutional neural networks. The circuits we designed in chapter 5 conform with the proposed archetypal accelerator in this section, in the sense that they have parameter, input and output stream interfaces. Conforming to this model, helps in deploying the designed accelerators in the server cluster environment.

In our proposed NAA deployment model, we allow computations to be distributed across several NAA compute nodes. This may be viewed as partitioning a streaming dataflow graph into multiple sub-graphs and deploy those individual sub-graphs on multiple NAA compute nodes. The resulting sub-graphs, then, need to communicate over a local area network. Figure 3.12 serves to illustrate the concept of spreading a directed acyclic dataflow graph (DAG) on multiple NAA compute nodes using an example. The illustrated DAG, in figure 3.12, represents a MapReduce job that consists of a stage of map tasks followed by an-

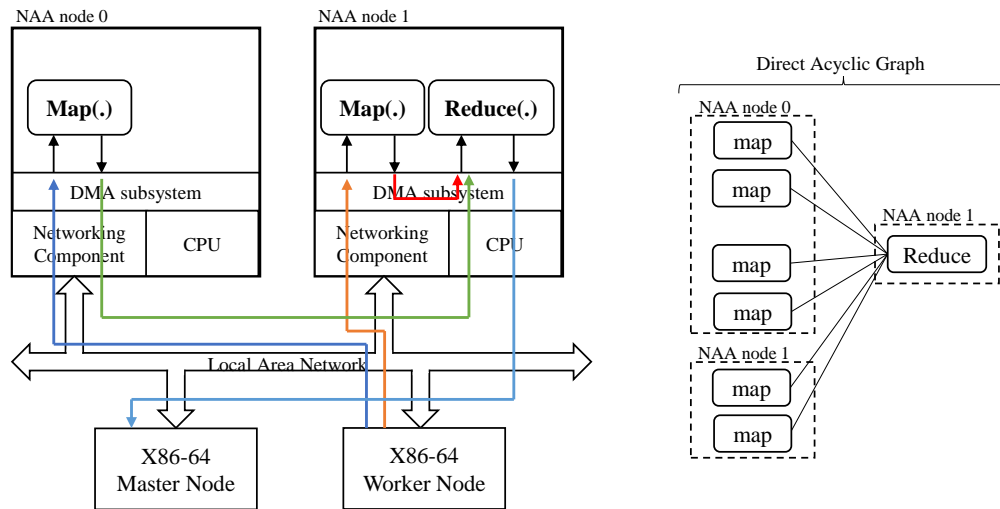


Figure 3.12: NAA computational pipeline example.

other stage of reduce tasks. Accelerators for the map tasks are deployed on both NAA node 0 and NAA node 1. An accelerator for the reduce task is deployed in NAA node 1 alongside another map accelerator. The flow of data between the accelerators is represented using colored arrowed lines (c.f. figure 3.12). Some of those lines represent data flowing between accelerators sitting on the same node, while other lines represent data flowing between different accelerators sitting on different nodes.

The flows of data between accelerators, whether they live on the same node or on different nodes, is managed jointly by the cluster’s master node and by the firmware instances running on the different NAA compute nodes. In order to manage those data flows properly, we introduce two different abstractions: (1) *Network streams*, and (2) *NAA programs*. *Network streams* are objects that the master node and the firmware instances maintain to exert control over a certain flow of data. They may be viewed as tunnels through which a certain data stream can travel across a local area network. Both the master node and the different firmware instances may dynamically allocate and deallocate *Network streams* in

order to flexibly establish communication channels between the different accelerators that are sitting on different NAA compute nodes. The Network stream class has multiple data members; the most important of these members are the *Network stream ID* , *source IP address*, *source port number*, *destination IP address*, and *destination port number*. The source, and destination information in a *Network stream* object allows the NAA compute nodes to properly direct the flow of data between the different accelerators across the NAA-augmented cluster.

Every NAA compute node maintains its own internal routing table that controls the flow of data internally within the NAA node. Every entry in the internal routing table consists of multiple fields; the most important of those fields are: (1) *source interface*, (2) *destination interface*, (3) *length*, and (4) *repetition*. The *source interface* can either be an accelerator's output interface, a network stream, or a pre-allocated buffer in memory. Similarly, the *destination interface* can either be an accelerator's input interface, a network stream, or a pre-allocated buffer that is populated with useful data. The *length* field indicates the number of bytes that will move across the designated route before it expires. The *repetition* field is used when the source interface is a pre-allocated buffer, and the destination interface is an accelerator's input interface or a network stream, in which case the data in the pre-allocated buffer is read into the destination interface N times, where N is equal to *repetition*.

The *NAA program* is a class of objects that is used by the NAA driver program to control the internal routing tables of all the NAA compute nodes in a certain NAA-augmented server cluster. Consequently, the *NAA program* object consists of the union of all the desired NAA node's internal routing tables. In summary, to build a computational pipeline, the NAA driver program needs to fetch the correct bitstream files from the bitstream repository, program the allocated NAA

compute nodes, define the necessary Network Streams, build a suitable *NAA program*, and finally send the corresponding portions of the *NAA program* to every NAA compute node.

3.3.5 Firmware Architecture

As mentioned earlier, the NAA driver program is responsible for orchestrating the execution of computational tasks on the NAA compute nodes. Initially, when NAA nodes are connected to the cluster's network, each NAA node starts sending periodic heartbeat signals to the *Resource manager*. Each heartbeat signal contains enough information on the NAA compute node to uniquely identify it. When the *Resource manager* receives a heartbeat signal, it adds the corresponding NAA node to a list of available nodes. To gain access to NAA compute nodes, the *Driver process* contacts the *Resource manager* and requests a pre-defined number of NAA compute nodes. The *Resource manager*, then, allocates the requested number of NAA compute nodes to the *Driver process*. Information about the allocated NAA nodes, such as IP address, FPGA model and memory capacity, are sent to the *Driver process* by the *Resource Manager*; The allocated NAA nodes may, as well, be informed about this handover. The *Driver process*, then contacts the *Bitstream repository* to fetch suitable FPGA bitstreams, and programs the FPGA devices on the allocated NAA nodes.

To instigate a computation, the *Scheduler* part of the NAA *Driver process* builds a suitable *NAA program* and disseminates it to the NAA compute nodes. A monitor process running on every NAA compute node, receives the *NAA program* from the driver, and acts upon it. We will refer to this monitor process as the NAA-side firmware. In addition to interpreting the *NAA program*, the NAA-side firmware acts as a mediator between the x86-64 server nodes and the

NAA node; It allows the *Driver process* or the *Resource manager* to control the low-level aspects of NAA nodes such as configuring the FPGA fabric or allocating/deallocating local buffers. On the server side, a middleware software layer implements all the necessary components, we previously mentioned such as the *Resource manager*, the *Driver process*, *Bitstream repository*, and the *Scheduler*. In addition to the middleware layer, a server-side firmware equivalent to the NAA-side firmware is implemented. The purpose of the Server-side firmware is to mediate between the middleware software layer instances running on the x86-64 servers and the NAA-side firmware instances running on the NAA compute nodes.

Layered Architecture

Our proposed NAA cluster system exhibits a layered architecture. Figure 3.13 illustrates the different layers involved. An *Application layer* sits at the top of the stack. The *Middleware layer* consists of all the services provided by the NAA driver program, the *Resource manager*, the *Bitstream repository*, and the *Scheduler*. Finally, at the bottom of the stack, an *NAA layer* is employed to manages all the low-level interactions with the NAA compute Nodes. The *NAA layer* is responsible for programming the FPGAs, delivering an *NAA program* to the NAA compute nodes, and feeding or receiving streams of data to or from the NAA nodes. In this work, we will limit the scope of our discussion to the structure and role of the *NAA layer*.

As shown in figure 3.13, the *NAA layer* consists of Server-side and NAA-side firmware instances. An NAA-side firmware instance is a piece of software that runs on the multi-core processor component of the NAA compute node. It implements the concept of internal routing tables, which are populated using

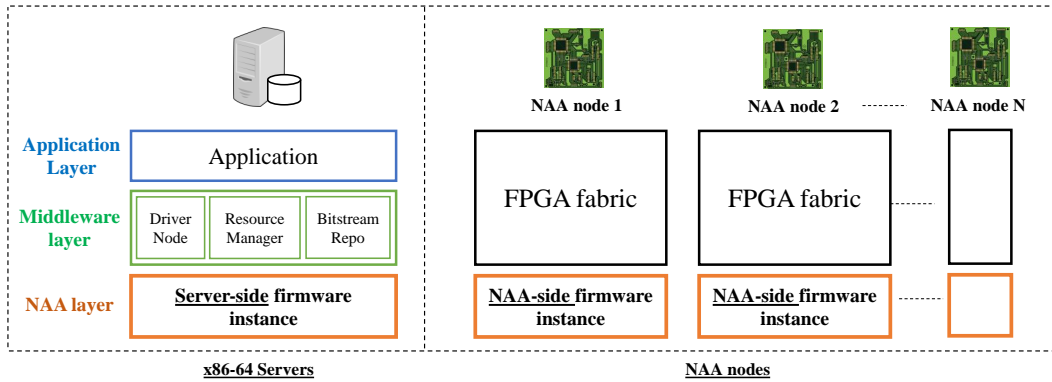


Figure 3.13: Layered Architecture

NAA programs, and the concept of *Network Streams* described in section 3.3.4. Furthermore, the firmware does all the necessary book keeping regarding the deployed FPGA-based accelerators, and reports status updates to the *Driver process* through heartbeats. The Server-side firmware provides a standard interface through which the *Driver process* or the *Resource Manager* can communicate with the NAA-side firmware instances and with the FPGA-based accelerators. The Server-side and the NAA-side firmware instances may employ the standard TCP/IP protocol stack to communicate, in which case the firmware implements a lightweight version of the TCP/IP protocol stack. The firmware provides a set of NAA commands through which the *Driver process* or the *Resource Manager* can control the internal aspects of the NAA compute nodes. In the following section, we will describe the necessary NAA commands made available by the firmware.

NAA Commands

An NAA-side server process, listening at TCP port 7 and running on the NAA's processor component is responsible for receiving NAA commands and other control information. When the NAA server process receives a piece of data, it for-

wards the data to another internal process, called Command interpreter. The Command interpreter can be in one of three states at a time: (1) *Listening*, (2) *Receiving*, and (3) *Executing*. Initially, the Command interpreter is in the *Listening* state, in which case, the interpreter waits until it receives a valid NAA command from the NAA server process. When a valid command is received, the interpreter can go to one of the two other states i.e., *Receiving* or *Executing*. If the received command requires additional data, the interpreter goes to the *Receiving* state; otherwise, it goes directly to the *Executing* state. In the *Receiving* state, the interpreter waits until the additional data is received completely before going to the *Executing* state. Finally, in the *Executing* state, the firmware executes the received command. When the received command is executed, the interpreter goes back to the *Listening* state.

Upon scheduling a computation on an NAA compute node, the *Driver process*, fetches the necessary bitstream from the *Bitstream repository* and tries to program the FPGA device on the corresponding NAA compute node. To program an FPGA device, the *Driver process*, first, attempts to establish a reliable TCP connection with the hosting NAA node at port 7 to send NAA commands. After establishing the reliable TCP connection with the NAA node, the *Driver process*, through its server-side firmware instance, sends a **program_bitstream** command to the NAA-side firmware instance followed by the desired bitstream file and an Accelerator Meta-data file. The Accelerator Meta-data file contains information about the bitstream file such as the accelerators involved, along with information about each individual accelerator. The most important pieces of information carried by the Meta-data file are the accelerators' ID numbers, the input/output stream interfaces' ID numbers, and the number, sizes and types of the dedicated buffers allocated for the accelerators (refer to section 3.3.4 for infor-

mation on the dedicated buffers). The firmware maintains a data structure that represents all the information about the accelerators; we call this data structure Accelerator Control Block (ACB). Upon receiving the Bitstream and the meta-data files, the firmware, then, programs the FPGA device using the Bitstream file, and uses the information in the meta-data file to allocate the necessary buffers, and to create the ACB.

After programming the NAA's FPGA device, the scheduler component of the *Driver process* might decide to buffer data on the NAA's On-board memory. In this case, the *Driver process* may instruct the NAA-side firmware to allocate a memory buffer on the NAA's main memory. To allocate a buffer on the NAA side, the *Driver process* sends an **allocate_buffer** command followed by a few parameters that determines the buffer's ID number, size and type. Buffering may be employed in cases where a working dataset might be used multiple times. Similarly, the *Driver process* may instruct the NAA-side firmware to deallocate a buffer using the **deallocate_buffer** command.

When the *Driver process* programs all its allocated NAA compute nodes, it attempts to establish a computational pipeline. As described in section 3.3.4, two important concepts may be used to establish this pipeline: (1) *Network streams*, and (2) *NAA programs*. *Network streams* are communication tunnels across which data may travel between NAA compute nodes. When it comes to allocating *Network streams* on NAA compute nodes, there are two types of *Network streams*: inbound and outbound. When an outbound *Network stream* is allocated on an NAA compute node, the node may actively instantiate and establish a connection with another node through this stream. On the other hand, if an inbound *Network stream* is allocated on an NAA node, the node may only passively receive a connection request from another node through this stream.

Consequently, for every inbound *Network stream* in one NAA node, there should be at least one outbound *Network stream* on another NAA node. Our firmware architecture supports a set of NAA commands for allocating both inbound and outbound *Network streams*. To allocate an inbound *Network stream*, the *Driver process* may use the **allocate_inbound_stream** command followed by a list of parameters that consists of the *Network streams's* ID, and the destination port number. Upon receiving this command, the NAA-side firmware launches a TCP server process that listens for connection requests at the TCP port number specified in the received command's parameters. Similarly, to allocate an outbound *Network stream*, the **allocate_outbound_stream** command may be used. When an outbound *Network stream* is allocated, the NAA-side firmware launches a TCP client process that immediately starts attempting to establish a connection with another server process that corresponds to another inbound *Network stream* on another NAA node.

After allocating the necessary *Network streams*, the scheduler function, on the *Driver process*, prepares a suitable *NAA program*. In section 3.3.4, we defined the *NAA program* as the union of all the desired internal routing tables in the allocated NAA nodes. When the *scheduler* is done with preparing the *NAA program*, the *Driver process* disseminates the portions of the *NAA program* to their corresponding NAA nodes. The **load_naa_program** command is employed to load the individual portions of the NAA program to the different NAA nodes. The server-side firmware instances, allow the NAA nodes to interact with the server nodes as if the servers are regular NAA nodes as well. Consequently, the *Network streams* may also be allocated on the server nodes, and the same is true for the *NAA programs*. The server-side firmware instance allows the regular server nodes, the worker nodes in this case, to emulate NAA nodes. This

emulation.

Finally, the *Driver process* may ask an NAA node to provide a summary report on all its internal aspects such as the deployed accelerators, memory usage, etc... To instruct the NAA node to send a summary report the **show_summary** command may be used.

Emulated NAA nodes

As mentioned earlier, the server-side firmware, which is installed on all the server nodes in an NAA-augmented server cluster, allows the master node and the slave server nodes (worker nodes in Spark's terminology) to emulate NAA compute nodes. From the perspective of the NAA-side firmware, all the compute nodes in the cluster, including the x86-64 server nodes, are NAA compute nodes. This perspective allows the NAA compute nodes to use the same protocols we previously described, such as Network streams and NAA programs, to communicate with the regular x86-64 server nodes. Consequently, outbound Network streams may be allocated on the worker nodes from which data may be fetched or to which data may be written; on the NAA-side firmware instances matching inbound Network streams should be allocated. To properly establish a compute pipeline, the *Driver process* should issue NAA commands in this exact order: (1) program the allocated NAA compute nodes using the **program_bitstream** command, (2) allocate the necessary buffers using the **allocate_buffer** command, (3) allocate all the necessary inbound *Network streams* on the NAA nodes using the **allocate_inbound_stream** command, (4) allocate all the necessary outbound NAA *Network streams* on the NAA nodes using the **allocate_outbound_stream** command, (5) allocate outbound *Network streams* on the emulated NAA compute nodes, which are running on server-side firmware instances i.e., on the master

and worker nodes, (6) disseminate the portions of the NAA program to the corresponding NAA compute nodes using the `load_naa_program` command.

Integration with Spark

Our proposed NAA system architecture exhibits a similar structure to that of a Spark cluster. In the Spark cluster environment, a driver node orchestrates the distributed execution of Spark tasks on multiple worker nodes; moreover, a cluster resource manager node is responsible for allocating compute and memory resources for Spark applications. Similarly, our NAA architecture have an NAA master node, an NAA cluster manager, and several NAA compute nodes (c.f. figure 3.8). Our proposed architecture may co-exist with Spark’s infrastructure; for instance, we can have a single resource manager node running both a Spark resource manager and an NAA resource manager. The driver node may also run a Spark Driver program along with an NAA driver program.

Another way of integrating our proposed NAA system architecture into the Spark environment, consists of modifying the Spark middleware to support NAA compute nodes. Figure 3.14 illustrates Spark’s layered architecture (on the left) and how it may be modified to add NAA functionality (on the right). The Spark layered architecture is divided into three layers: (1) The Application layer, (2) the Spark core layer, and (3) the resource manager layer. The Spark core and resource manager layers, together, form the Spark middleware layer. In the application layer, many application libraries that use the underlying Spark middleware exist such as Spark SQL, ML-lib, and GraphX. The scheduler component in the Spark core is responsible for scheduling application tasks on the worker nodes. To add NAA functionality to a Spark cluster, the Spark core may be modified to add an NAA scheduler component, which is responsible for the decision process regarding

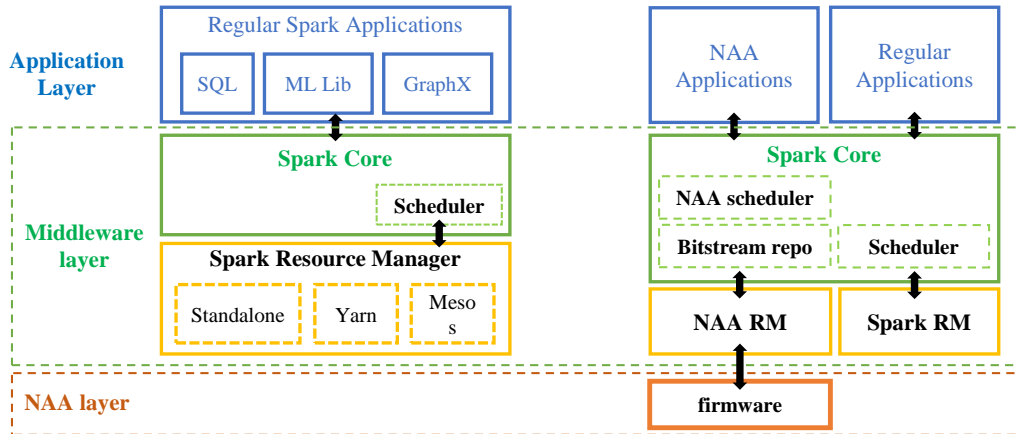


Figure 3.14: Integrating our NAA system architecture into Spark

the allocation of Network streams and the composition of NAA programs. A bitstream repository component may also be added to the modified Spark core, since deciding on the right FPGA bitstreams in an NAA cluster should be part of the scheduling process in the NAA-augmented server cluster. Finally, the previously described NAA layer is added at the bottom of the modified Spark layered architecture to allow the server node to control with the NAA nodes through NAA commands.

In addition to including the NAA scheduler and the Bitstream repository components, the modified Spark may also add new Spark transformations and objects such as a `mapFpga(path)` transformation and a Hardware RDD class. The `mapFpga(path)` transformation, when invoked on a Spark RDD, instructs the framework to allocate several NAA compute nodes for the transformation. The `path` parameter indicates the location of the Bitstream and Accelerator Metadata files that should be involved in the computation. The modified framework, through the NAA- and Server- side firmware instances, programs the allocated FPGA nodes. The scheduler components, also through the firmware, allocates the necessary *Network streams*, composes a suitable NAA program, and disseminates

the different portions of the *NAA programs* to their corresponding NAA compute nodes. Hardware RDD objects keep track of the data buffered on the NAA compute nodes. The framework translates all operations on hardware RDDs into suitable combinations of *Network streams* and *NAA programs*.

3.4 Experimental Evaluation

The NAA system architecture proposed in section 3.3.2 along with the firmware architecture proposed in section 3.3.5 can be used to deploy custom designed accelerators in the Spark data center environment. In order to validate this assumption, we developed an experimental setup that consists of an x86-64 server machine and one NAA compute node. The NAA compute node, as defined in section 3.3.2, is a single board computer centered around an FPGA device. The closest FPGA-based computing platforms to the concept the NAA compute node are SoC-based FPGA development boards with Network Interface Cards and relatively large on-board RAM memory.

3.4.1 NAA Platform

We employed a ZedBoard [35] platform to implement our NAA compute node. The ZedBoard is an Evaluation and Development board from Avnet that features the Xilinx Zynq 7020 SoC device. The Zynq SoC device brings together (1) an FPGA device and (2) a dual-core ARM processor component on the same package. The ZedBoard also features many other components such as a 1Gbit Ethernet PHYceiver, an SD card reader, and 512 Mbytes of DDR III RAM memory.

3.4.2 Firmware Implementation

The NAA-side Firmware is implemented as a bare-metal software running on the dual-core ARM processing system of the Zynq device. A lightweight and open source implementation of the TCP/IP protocol stack, called LWIP [86], is employed to provide reliable and robust communication channels with the server nodes. LWIP allows our NAA compute node to flexibly initiate TCP connections with any other entity in the network, and to receive and accept a connection request from any other compute node in the same network. Moreover, LWIP allows the NAA node to send and receive UDP segments. Network streams are implemented as TCP sockets, and the internal routing table is implemented as an array of C structures with ID, source, and destination fields. The Dynamic reconfiguration of the Zynq FPGA fabric is accomplished through the Processor Configuration Access Port (PCAP).

3.4.3 Experimental Setup

Our experimental setup consists of an x86-64 server node, an Ethernet switch, and one ZedBoard that implements an NAA compute node. A Java process that emulates an NAA compute node on the x86-64 server node was deployed and a preliminary implementation of the NAA-side firmware was also deployed on the ZedBoard's ARM processor component. Figure 3.15 illustrates our experimental setup. To test the concepts of *Network streams* and *NAA programs*, we prepare an FPGA-based accelerator that performs matrix-vector multiplications. The accelerator has two stream interfaces: (1) an input interface through which the input vector is received by the accelerator, and (2) an output interface through which the resulting vector is received from the accelerator.

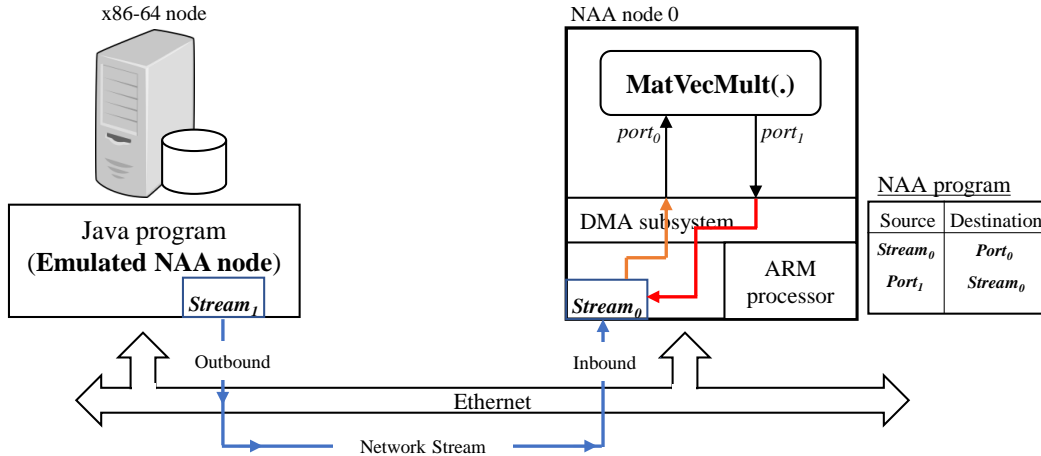


Figure 3.15: Experimental Setup

The Java process, running on the x86-64 server, allocates a single inbound *Network Stream* at the NAA-side firmware instance and attempts to open a TCP connection with the server process that corresponds to it. In this experiment, we manually composed the *NAA program*; the NAA program is shown on the right side of figure 3.15. The Java process, then, sends the NAA program to the ZedBoard, resolving its internal routing scheme (c.f. figure 3.15). After establishing the TCP connection, the server node starts sending input vectors through the allocated Network stream to the NAA compute node. The output of the matrix-vector operation is, then, sent back to the server through the same Network stream. The setting was tested with different vectors and was able to correctly compute the results.

3.5 Conclusion

In this work, we proposed a Network-Attached Accelerator (NAA) system architecture for deploying FPGA-based accelerators in a server cluster data center

environment. We defined an NAA node as a standalone compute node centered around an FPGA device. The NAA node contains enough intelligence and computing power to work independently, and to provide computation services to other computer nodes in the network. We developed a Firmware architecture that allows legacy cluster computing frameworks such as Spark to invoke the NAA nodes and to use them to build complex compute pipelines and run tasks on the established pipeline.

Chapter 4

Accelerating Convolutional Neural Network Operations in the Spark Data Center Environment

Deep Learning has received a lot of attention lately due to the record-breaking results it is able to achieve in several machine learning applications such as computer vision, speech recognition, and natural language processing. Traditionally, practical machine learning algorithms, such as classification and pattern recognition algorithms, relied on a good representation of the input training data to construct a model for the observed data; this model would then be used to recognize future unseen data patterns during the inference phase. To achieve a good representation of the data, fixed feature extractors are used. Typically, a fixed feature extractor transforms the raw input into a suitable representation. Representation and Feature extraction are fundamental issues in machine learning because a poor representation of the input data, can undermine the performance of even the best machine learning algorithms [7]. Therefore, feature engineers craft fixed domain-specific feature extractors that can extract high level representations from the raw input data and feed these representations to a machine

learning model, such as a classifier, for it to train on.

Deep learning, on the other hand, is a novel approach to machine learning where complex representations, suitable for the problem in hand, can be automatically inferred during the training process. Deep Learning algorithms, develop a multi-layered hierarchy of representations, where higher-level abstract features are extracted from lower-level more concrete features [7,8]. Those representations are not fixed nor pre-designed as in traditional machine learning approaches but are rather learned during the training phase.

The most successful deep learning algorithms are those based on deep Feed-forward Artificial Neural Networks, specifically deep multi-layer convolutional neural networks. Deep Convolutional neural networks were inspired by the structure and function of the visual cortex in the human brain [12]. The main distinguishing feature of these networks is that they incorporate convolutional layers in their structure. Empirical studies have shown that Convolutional neural networks outperform shallow neural networks in many applications such as character and object recognition and can even achieve human-like or even superhuman performances in some applications [8]. For instance, LeNet 5, a pioneering convolutional neural network-based classifier for handwritten character recognition was able achieve a classification accuracy of 99.87% [3]. In image and object recognition, GoogLeNet [30] won the 2014 ILSVRC [87] competition, achieving the best mean average precision and the lowest classification error on the ImageNet dataset [88]. Lately, convolutional neural networks submitted to the ILSVRC competition are capable of classifying objects that even humans find trouble in classifying.

Although, the concept of deep convolutional neural networks first appeared in the late 1960s, it wasn't until recently that it achieved astounding results

and grabbed the interest of the academic and industrial communities. The unprecedented computing power that current hardware platforms can provide made training over large datasets possible and gave the above concept a chance to shine because convolutional neural networks and Deep learning algorithms in general are quite beneficial only when trained over large trails of data [7]. Also, as we will describe in section 4.1, the algorithmic complexity of the convolutional layer presents a real computational bottleneck in both training and inference phases, so to mitigate the increasing complexity of deep convolutional neural networks, the training algorithm is usually distributed across a computer cluster. For instance, in a study conducted at Google [48], a 1 billion-connection neural network model was trained on a cluster with 1000 16-core machines. The demand for high performance in convolutional neural networks, led researchers to accelerate the training algorithms of deep neural networks in two ways: (1) by distributing the training workload across a shared-nothing cluster architecture (Data Parallelism) [48, 49] and (2) by coupling the main processor with a domain-specific programmable co-processor, such as a Graphical Processing Unit (GPU) or a Field Programmable Gates Array (FPGA), that helps the main processor by offloading the computationally intensive kernels into the core. Recently, there has been a surge of interest in combining both approaches to scale up the performance of deep convolutional neural networks [17].

In this work, we combined both approaches. we proposed a custom-designed FPGA-based 2D convolution filter that can be used to accelerate the computation of the multi-layer convolution operator used in the distributed training of convolutional neural networks. The design of the accelerator was inspired by Farabet [89] and was upgraded with 2 circuit design optimizations; (1) A re-timing transformation was applied in order to increase the clock frequency of

the circuit and hence its performance, and (2) a feed-back loop was applied to compute the multi-layer convolution without the use of an accumulator. According to this work, the optimized FPGA-based accelerator (filter) is intended to be used as a reconfigurable component that provides FPGA-based acceleration for machine learning and data analytic applications. Additionally, we aimed to extend the Spark [29] environment to be able to target the introduced system architecture. In short, FPGA-based accelerators should run in the data center under the Spark environment in order to provide a better performance per Joule than a pure distributed software implementation.

The remainder of this chapter is organized as follows. Section 4.1 provides a brief background on deep convolutional neural networks. In Section 4.2, we discussed the related work in this area. Section 4.3 presents our system overview, in addition to the custom FPGA-based accelerator design. In section 4.4, we describe the experimental setup and report/discuss the resulting speedup and energy savings due to the FPGA-acceleration. Finally, we conclude our work in section 4.5.

4.1 The Multi-layer Convolution Operation

Convolutional Neural networks are a special kind of deep feed-forward artificial neural networks in which the connectivity pattern is inspired by the structure of the mammalian visual cortex [12]. They consist of a series of stages or layers of computations where the first few stages are composed of convolutional and pooling layers. Neurons in a convolutional layer are organized into two-dimensional arrays, called feature maps, where every neuron in a feature map is connected only to local rectangular patches of neurons in the feature maps of the previous

layer through a set of trainable weights called filter banks [12]. All the neurons in one feature map share the same filter bank. This arrangement, also known as parameter sharing, provides an important characterizing feature of convolutional neural networks: translation invariance which states that if an image is translated a few pixels, a trained network can still recognition the same object within the picture [8]. Mathematically, the operation involved in the convolutional layer is nothing other than the discrete multilayer convolution shown in Algorithm 1. After, the convolution, a bias is added to every feature map before applying a point-wise non-linear activation function.

Algorithm 1: Multi-layer Convolution Operation

Input: An input three dimensional array $X(q, m, n)$, where $0 \leq q \leq Q$ and $0 \leq m < M$ and $0 \leq n < N$

Output: An output three dimensional array $Y(r, m, n)$

for $r = 0$, $r < R$, $r ++$ **do**

for $q = 0$, $q < Q$, $q ++$ **do**

for $m = 0$, $m < M$, $m ++$ **do**

for $n = 0$, $n < N$, $n ++$ **do**

for $k = 0$, $k < K$, $k ++$ **do**

for $l = 0$, $l < L$, $l ++$ **do**

$Y[r][m][n] += W[r][q][k][l] \times X[q][m+k][n+l];$

return $Y(r, m, n);$

A typical, convolutional neural network consists of several stages of convolutional and pooling layers before applying one or few stages of fully connected layers at the output [8]. Training deep convolutional networks is performed by defining an objective function which measures the error between the output of the network and the desired output. The objective function is then minimized using the Stochastic Gradient Descent Algorithm [12]. Both the training and inference algorithms of convolutional neural networks involve the multi-layer convolution

algorithm during the forward pass [8]. Notice that the complexity of Algorithm 1 is $O(RQMNKL)$, where M and N are the dimensions of the output feature maps, R and Q are respectively the number of input and output feature maps, and k and l are dummy variables. Given that deep neural networks are usually trained over large volumes of data [7], the complexity of the training can quickly grow to unmanageable levels. Therefore, in this work we investigate accelerating Algorithm 1 using a custom designed FPGA-based accelerator.

4.2 Related Work

As mentioned earlier, the complexity of training convolutional neural networks can be mitigated by distributing the training/inference workload across a cluster or by employing application domain-specific co-processors tailored for convolutional neural networks. Several approaches have been implemented in the literature. A popular approach, to accelerating neural network inference, is to train the network off-line using a normal multi-core system or a distributed platform, transfer the weights of the network to a custom-designed hardware accelerator tailored for neural network computations, and perform the inference on custom hardware.

In [89], an efficient implementation of convolutional neural networks on a low-end DSP oriented FPGA was investigated. The system used a single FPGA with an external memory module. A programmable Convolutional neural net processor (CNP) was designed; the processor consists of a 32-bit Soft Processor implemented in the FPGA reconfigurable fabric along with a Vector Arithmetic and Logic Unit (VALU). The VALU consisted of all the basic operations of convolutional neural networks including a 2D convolutional kernel based on the sliding

window architecture. The processor was designed for low-power embedded vision systems, where the neural network is first trained off-line on a traditional computer, and then uploaded to the CNP in order to accelerate the inference process. The design was able to process 10 frames/second.

A second-generation architecture of the CNP was proposed in [90] to increase the data throughput by adding multiple parallel Vector processing units and allowing them to seamlessly operate on individual streams of data between them. The architecture was implemented on a Xilinx Virtex 6 FPGA and was able to process more than 30 frames/second. In the same line of research Farabet et al. [27] proposed a scalable data-flow hardware architecture for general vision applications called NeuFlow along with a data-flow compiler for the architecture called LuaFlow.

A similar work to [27] was presented in [91], where a scalable, low-power co-processor was presented. The co-processor consists of an array of configurable processing elements that perform the operations required in the inference phase of convolutional neural networks. The system was designed for the real-time execution of these networks and is capable of achieving a peak performance of 227 G-ops/second on a Zynq chip with a Kintex-7 programmable logic.

In [25], Zhang et al. proposed a design scheme for exploring the design space of FPGA-based implementations of convolutional neural networks. It was shown that there could be 90% performance difference between 2 solutions with the same resources. The optimization also takes buffer management and memory bandwidth into consideration. As a case study, [25] implemented an optimized accelerator on a VC707 FPGA that achieves a peak performance of 61.62 G-flops under 100MHz clock frequency.

Incorporating FPGAs in accelerating the training of convolutional neural net-

works was given little attention in the literature. In fact, the training of large networks could take days and even weeks of training [17, 48]. An approach for accelerating both the training and inference of convolutional neural networks using GPUs was proposed by Strigl et al. [74]. In order to make the access pattern more linear, the “unfolding” technique was used, where the input feature map of each convolutional layer would first be copied to a matrix, where the elements of each rectangular patch in a feature map are organized into a row in the matrix. Once this is done, the training and inference processes can be implemented as matrix operations using a GPU. The benchmarks were performed on a platform with an Intel Core i7 860 processor and a GeForce GTX 275 GPU running Linux Ubuntu. The GPU implementation of the LeNet5 network was found to give 2 to 8 times speedup over an optimized pure software implementation, that takes full advantage of the SSE extensions of the CPU.

The excellent performance of GPUs, in addition to their flexibility, led to the adoption of GPUs as accelerators for deep neural networks, for instance deep neural network libraries such as Caffe [92, 93], Torch7 [94], Theano [95] and TensorFlow [96] employ GPUs for accelerating the training of deep networks. Although GPUs are very attractive for software developers, custom architectures that are tailored for a certain application domain can deliver a two orders of magnitude improvement in the energy efficiency (G-ops/Watt) [27]. DSP-oriented FPGAs can be used to implement custom designs; however, they lack support for efficient floating point arithmetics. Consequently, most of the designs implemented on FPGAs use fixed-point arithmetics.

Gupta et al. [97] investigated the effect of limited precision of fixed-point data representation and arithmetics on the training of deep convolutional neural networks. It was found that the precision required to train a neural network

depends on the rounding mechanism and on the neural network architecture. LeNet-5 was successfully trained using the MNIST dataset with a Q4.12 precision scheme (4 bits for the integer part and 12 for the fractional part) given that the stochastic rounding mechanism is used.

The other approach to accelerate the training of deep networks is the parallelization of the training procedure. It is however critical when distributing the training process, to decide on the distribution scheme. In [49], Zinkevich et al. proposed and described the Parallel Stochastic Gradient Descent Algorithm (PSGD). In this approach, data parallelism is employed. In other words, the training dataset is distributed across multiple computing nodes. Each node, having access to its own local data, trains a local copy of the neural network model. Then a master subroutine collects all the models and aggregates them into a single one by averaging their weights. PSGD is MapReduce friendly since no communication overhead between the nodes is incurred during the training.

Another approach to distributing the training is achieved through model parallelism across a single machine and across a cluster. The work in [17], developed a software framework for distributing the training of deep networks called DistBelief. The framework allows the user to partition very large models across several machines. The work also concluded that MapReduce is typically ill-suited for iterative computations such as those used in convolutional neural networks.

In an effort to bridge the gap between distributing the training of deep convolutional neural networks and FPGA acceleration, the work in [98], proposed an FPGA-accelerated Hadoop cluster architecture. Also, the Parallel Stochastic Gradient Descent algorithm was employed in the context of Hadoop MapReduce to accelerate the training of deep convolutional neural networks.

Similarly, to the work in [98], the current paper introduces a system architec-

ture that involves Spark as a parallel programming framework, and FPGA-based acceleration to scale up the performance of the distributed tasks assigned through Spark.

4.3 Fpga-based accelerator for convolutional networks

As mentioned earlier, the performance of deep learning and heavy data analytic workloads can be improved by employing two methods: (1) distributing the data and the related tasks to a number of disparate computing systems, and (2) employing application domain specific cores such as GPUs and FPGAs. In this work, we employ loosely-coupled FPGA-augmented compute nodes, arranged in a cluster configuration to accelerate the multilayer convolution for convolutional neural networks.

4.3.1 System Overview

Figure 4.1 shows the functional architecture of our system. The system consists of FPGA-augmented computing nodes arranged in a shared-nothing computer cluster architecture. Each computing node is equipped with a Xilinx Zynq-7020 SoC which contains a dual core ARM Cortex-A9 processing system coupled with an FPGA, and uses external DDRIII DRAM for external storage, an SD non-volatile memory storage device for storing files, and an Ethernet controller to provide network connectivity.

The cluster is managed by a middle-ware software suite that consists of the parallel programming framework, Spark [29], which provides a simple interface for

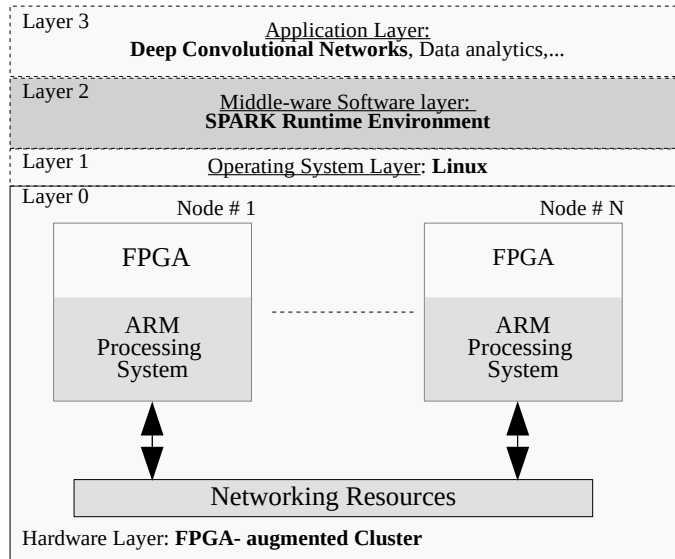


Figure 4.1: System Overview.

programming the cluster. In contrast to the work in [89], [90] and [27], which were designed to provide real-time processing for specific applications, our system was designed with the goal of providing reconfigurable-based acceleration for common data analytic workloads in the data center. In this work, the Spark runtime environment was extended to provide seamless integration of the FPGA-based accelerators into the cluster. A custom designed FPGA-based accelerator for the 2D multi-layer convolution was implemented in the FPGA. In the following, section we describe the architecture of our accelerator.

4.3.2 Hardware Architecture

The multi-layer convolution is the major performance bottleneck in the training/inference of deep convolutional neural networks. Interestingly, the software implementation of convolutional networks was evaluated and profiled in [98] for performance bottlenecks and was found to consume around 90% of the total

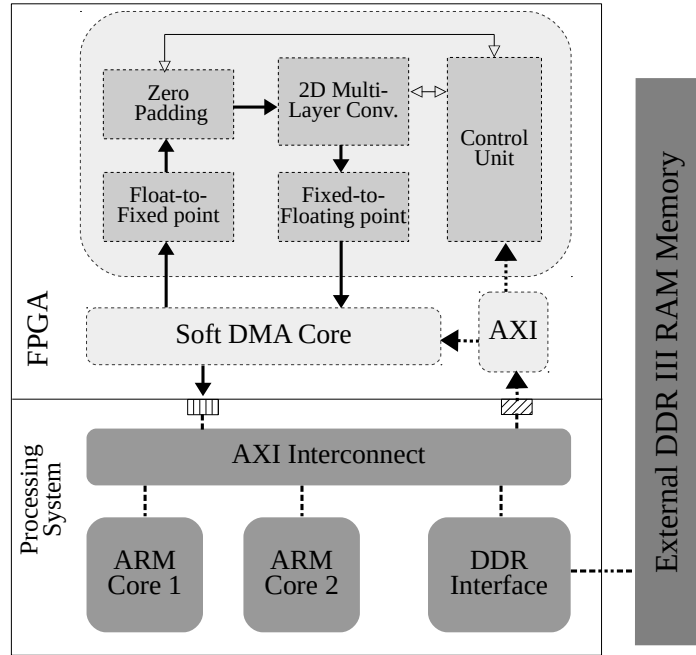


Figure 4.2: Functional Architecture of our compute node.

computation time. Accordingly, enhancing the performance of the multi-layer convolution is critical to improving the performance of the training/inference of convolutional neural networks.

In this work, we aimed at designing an accelerator that would perform the multi-layer convolution. The accelerator was designed to be (1) more performant than software implementations and (2) more energy efficient. Figure 4.2 shows the architecture of our accelerator. It contains a floating-to-fixed-point conversion circuit, a zero-padding circuit, a Pipelined 2D convolution filter, a fixed-to-floating-point conversion circuit, and finally, a Control Unit.

In our design, we employed a variation of the 2D multi-layer convolution filter used in [89], with 2 main optimizations: (1) A re-timing transformation technique, used to reduce the clock period of the circuit and thus to increase the performance of our system and (2) a feedback loop, devised to compute the multi-

layer convolution without the need for an accumulator circuit after the output of the convolution filter.

Figure 4.3, shows our optimized architecture of the 2D multi-layer convolution filter. As in [89], the circuit consists of several 1D Finite Impulse Response filters (shown in figure 4.3), that operate on the input stream in parallel. For completeness, we briefly describe the functionality of the 2D FIR circuit. Each 1D FIR filter corresponds to one row in the convolution mask. Also, as depicted in figure 4.4 each 1D FIR has 2 input ports: A and B. The images that need to be convolved are sent to the filter, line after line through the input port of the 2D convolution circuit, which is connected to the input port A of each 1D FIR filter. The output of each 1D FIR filter is delayed for N_W cycles, where N_W is the size of our image's line, before feeding it back to the 1D FIR that corresponds to the next row of the convolution mask. Shifting the values of the outputs of the 1D FIR filters through the delay lines corresponds to shifting the convolution mask over the input image. The aforementioned architecture is known as the sliding window architecture.

In [89], the input port has a high fan out, which severely limits the clock frequency and hence the performance of the circuit. In our design a re-timing transformation was applied to the filter by adding pipeline latches. In each 1D FIR filter, shown in figure 4.4, a pipeline latch was added between each multiplier and adder in order to reduce the critical path of the circuit. As shown in figure 4.3 and 4.4 another set of pipeline latches was added to reduce the high fan out of the input port (the structures enclosed in red boxes in both figure 4.3 and 4.4). The resulting circuit is a $(K_W + K_H - 3)$ -level pipelined system with a very low critical path, where K_W and K_H are, respectively, the width and height of the convolution mask.

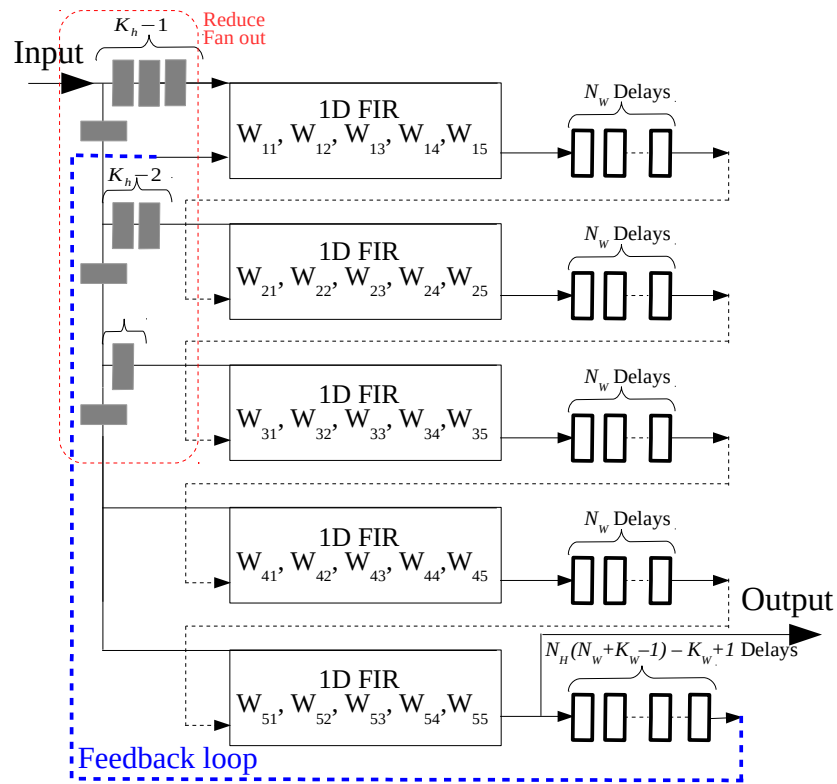


Figure 4.3: Optimized architecture of the multi-layer convolution.

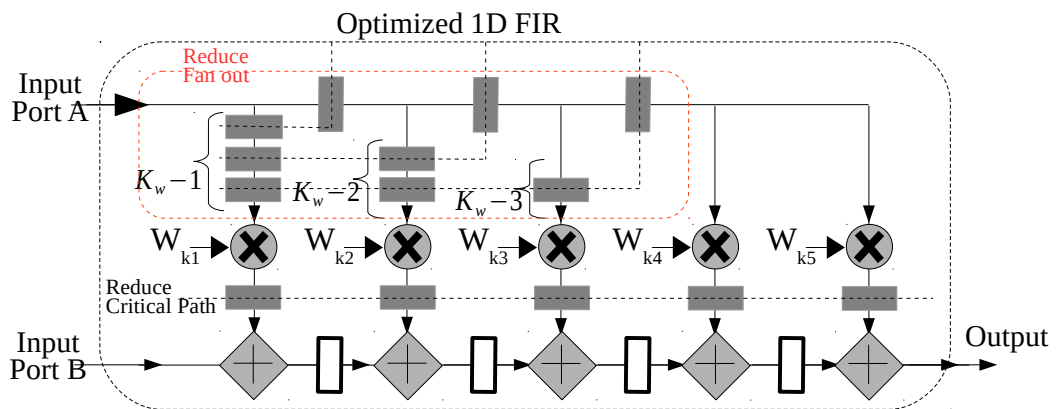


Figure 4.4: Optimized 1D-FIR filter.

In order to compute the multi-channel convolution, the resulting images from the convolution filter must be accumulated. Typically, an accumulator circuit could be added at the output of the filter, as in [89]. Unlike [89], our approach was to delay the result of the 2D multi-layer convolution by $N_H \times (N_W + K_W - 1) - K_W + 1$ times and then to feed it back to the input of the 2D FIR filter. The resulting circuit performs convolution and accumulation without an accumulator at the output of the convolution filter.

The sliding window architecture requires padding zeros at the end of each line. The zero-padding circuit, shown in figure 4.2 is responsible for padding the required number of zero-valued samples to the end of each line and to the end of each image. Without this circuit the software running on the ARM core would have to pad the required zeros by writing them to the stream consuming additional energy.

As for the memory interface, A multi-channel Direct Memory Access controller is employed to read packets of 32-bit single-precision floating point numbers from memory and feed them as a pipelined stream to the accelerator. The floating-point numbers are first converted to fixed-point finite-precision numbers, which are fed to the 2D multi-layer convolution filter (refer to figure 4.2). The convolution filter processes the incoming stream and produces another output stream. The DMA controller then reads the resulting stream back to memory for the higher-layer software processes to consume.

Finally, our design can be configured for different sizes of input images by providing access to memory-mapped configuration registers in the control unit. The only fixed parameter in this design is the size of the convolution kernel, which can be changed through dynamic reconfiguration.

4.3.3 Software Layer

As described above, our system design is composed of two main layers: the hardware layer and the software/middle-ware layer. The hardware layer aims at designing an optimized convolution architecture to perform hardware-accelerated training of the convolutional neural network on local nodes. The second layer i.e. the Software/Middle-ware layer is responsible for distributing the convolution horizontally across several nodes in a data-center-like infrastructure, while vertically offloading the convolution on each node to the FPGA. To this end, we have augmented Spark middle-ware with custom transformations that offload the execution of tasks sent from the master/driver, particularly the convolution tasks, to FPGA accelerators. In simple words, we have created a new Spark operation called `mapFPGA` along with a newly designed RDD called `hardwareRDD`. In the same way that a `map` operator would instantiate a RDD of type `MappedRDD`, `mapFPGA` would create an instance of `hardwareRDD`. As any RDD, `hardwareRDD` is a subclass of the base class: `RDD`. The extension of the base class: `RDD`, was achieved by overriding the `compute` function of `RDD` base class in a way to load a native C library into Spark runtime. The native C library accepts two buffers for two 2-D arrays. The first buffer stores the image pixels in single precision floating point, while the second buffer stores the kernel parameters. The C library `mapFPGA` is called on a RDD of type `Array[Array[Float]]` i.e. an image (`RDD[Array[Array[Float]]]`). This RDD is in fact nothing but a distributed collection of gray scale images, with an iterator to loop over each partition of the RDD in a distributed manner.

Our current prototype supports distributed hardware-aided convolutions that ultimately fit into training large convolutional network models. Simply, given our initial prototype, when the driver program invokes `mapFPGA` operator on

an RDD of type `Array[Array[Float]]` implicitly, the distributed convolution will be offloaded to FPGA while still integrating seamlessly with spark runtime. While we are targeting mainly convolutional neural networks, the same system architecture can be used to accelerate other applications.

4.4 Methodology and experimental results

As discussed earlier, custom designed architectures tailored for specific applications can deliver better performance per energy than general purpose architectures such as multi-core processors. In order to validate this assumption in the context of deep convolutional neural networks, we evaluate the performance of our system, described in section 4.3, with the FPGA-accelerated multi-layer convolution kernel.

4.4.1 Experimental Setup

we employ ZedBoards [35] as our FPGA-accelerated compute nodes. The ZedBoard features the Xilinx XC7020 Zynq SoC device along with a 512 MB DDRIII RAM, an SD Card Reader, an Ethernet controller, and a set of complementary accessories. Each node is running a Linux operating system based on Linux kernel image 3.16. We have developed, implemented and validated the custom-designed FPGA-based multi-layer convolution accelerator detailed in section 4.3. Table 4.1, report the FPGA resource utilization of our accelerator.

4.4.2 Benchmark

A suitable benchmark to evaluate our accelerator is the LeNet 5 [99] convolutional neural network, useful for handwritten digit recognition. The size and depth

Table 4.1: Resource Utilization.

Resource	<i>DSP</i>	<i>Block RAM</i>	<i>Flip-Flops</i>	<i>LUT</i>
Used in the design	25	6.5	5667	4,193
Available on Chip	220	140	106,400	53200
Utilization	11.36%	4.64%	5.33%	7.88%

of LeNet 5 is manageable by the Zedboard. Therefore, they can be used as a tool to study, and validate our approach. The LeNet family involves multi-layer convolutions with a kernel of size 5x5. Typically, a large number of neural network architectures involve 5x5 kernels [100]. Although the above exhibited accelerator design approach, shown in section 4.3, tends to be generic, in the following scenarios we only employed 5x5 2D multi-layer convolution filters to administer our tests. In the following, we evaluated our design approach by performing 2 tests: (1) A performance test and (2) an energy efficiency test.

4.4.3 Speedups resulting from employing the FPGA-based accelerator

For the performance test, we measured the time required for a single FPGA computing node, that implements the circuit described in section 4.3.2, to compute a single output feature map of a multi-layer convolution. We measured this latency with different numbers of input feature maps and different feature map sizes, and we compared those measurements with the time required to do the same computation, using a pure software approach, on an ARM A9 processor core running at 667 MHz. Initially, we ran our FPGA circuit at 150 MHz; later, we varied

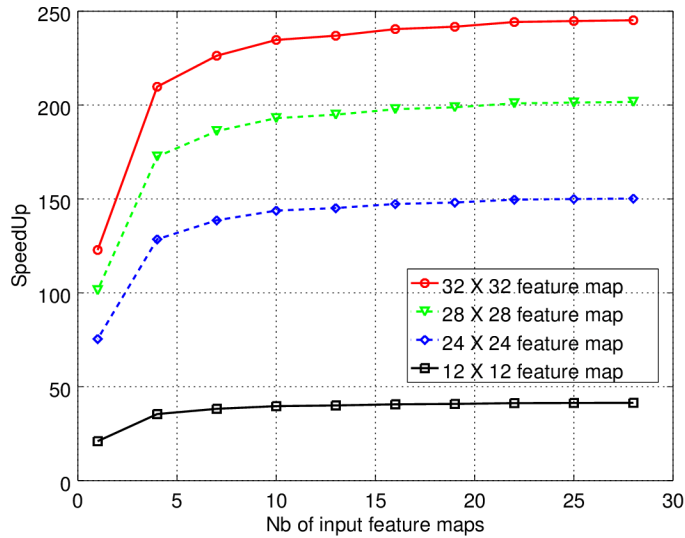


Figure 4.5: Speedup over an ARM core implementation as a function of the number of input feature maps

the operating frequency to study its effect on performance. We observed that the FPGA-based convolution circuit significantly outperforms the ARM A9 implementation of the multi-layer convolution. Here, we define the *speedup ratio* as the ratio of the computational latency of the processor core to that of the FPGA-based circuit. In figure 4.5, we vary the number of input feature maps, and the size of the input feature map, and we report the corresponding measured *speedup ratios*.

As shown in figure 4.5, the *speedup ratio* increases with the number of input feature maps, until it saturates and starts exhibiting a horizontal asymptotic behaviour. Launching transactions between external memory and the FPGA circuit incurs some overhead: To launch a computation on the FPGA circuit, a software running on the ARM processor core, invokes a driver which schedules a DMA transaction. The soft dma core, shown in figure 4.2, then, starts streaming the input feature maps from external memory to the hardware circuit. This

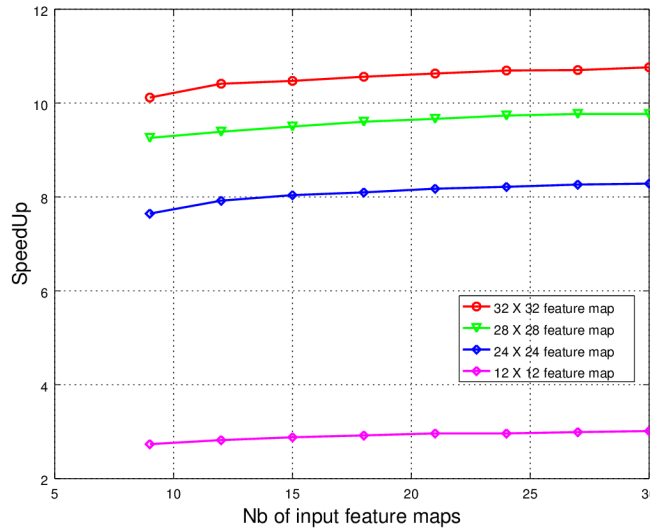


Figure 4.6: Speedup over a Core i7 CPU core implementation as a function of the number of input feature maps

overhead makes streaming small number of feature maps, or small feature maps inefficient. Transferring a large number of feature maps, may hide this overhead and allows the hardware accelerator to achieve its full potentials. This explains why the *speedup ratio* increases with the number of input feature maps, or with the size of the input feature map. This asymptotic limit reflects the achievable speedup for the current operating frequency. To make a fair comparison, We also measured the latency of a software implementation of the multi-layer convolution on a Core i7-4510U processor core running at 2 GHz. We found that the FPGA implementation can outperform the Core i7 processor core as well. Figure 4.6 shows that more than 10 times speedup over a Core i7 implementation may be achieved.

To further investigate the relationship between the asymptotic limit and the operating frequency, we measured the achievable speedup over different operating frequencies. Figure 4.7 shows the results of this investigation.

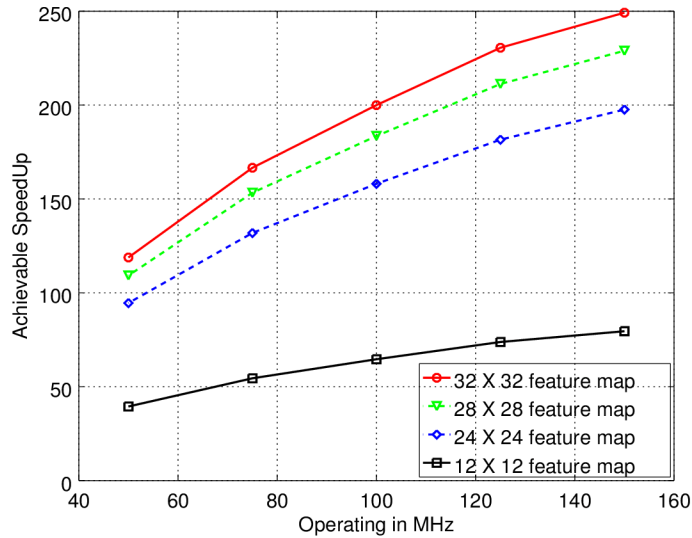


Figure 4.7: The achievable speedup as function of the operating frequency.

To sum up, we deployed the basic convolution circuit described in section 4.3.2 on an FPGA compute node. Although the circuit occupied less than 12% of the FPGA resources, it significantly outperformed a high-end Core i7 processor core by around 10 times. The remaining resources may be employed to further boost the performance of the multi-layer convolution; moreover, Spark’s extension allows the inference and the training to be distributed across a cluster of FPGA compute nodes, where each node takes part of the task-load. The raw computing power provided by the FPGA, coupled with the ability to distribute the computation across a cluster of FPGA compute nodes, makes this platform perfect for deploying convolutional neural networks in the Spark Data center environment.

4.4.4 Performance Model for the Multi-layer Convolution Operation

In order to further understand the speedup results shown in the previous section, we develop an empirical latency model for the Multi-layer Convolution Operation

on both FPGA and CPU platforms. Note that the hardware circuit, described in section 4.3.2, computes a single output feature map using I input features, in $(N + K - 1)^2 \times I \times \frac{1}{\beta_R}$ clock cycles. Where $N \times N$ is the size of the input feature map, I is the number of input feature maps, β_R is the read memory access speed in samples per second, and K is the size of the filter mask. The output of the multi-layer convolution is then written back to memory, the latency of writing the output feature map to memory is $M^2 \times \frac{1}{\beta_W}$, where M^2 is the size of the output feature map, and β_W is the write memory access speed. So the total latency of computing a single output feature map using the FPGA-based accelerator circuit shown in figure 4.3 is:

$$L_{FPGA,5 \times 5} = (N + K - 1)^2 \times I \times \frac{1}{\beta_R} + M^2 \times \frac{1}{\beta_W} \quad (4.1)$$

Note that $L_{FPGA,5 \times 5}$ grows linearly with I . In typical Convolutional Neural Networks I is on the order of hundreds of feature maps. When I is large, the computational latency, in microseconds, of computing one output feature map may be estimated by:

$$L_{FPGA,5 \times 5} \approx \frac{I \times (N + K - 1)^2}{\beta_R \times f_{clk}} \quad (\mu sec) \quad (4.2)$$

In figure 4.8(a), we varied the number of input feature maps I , and the size of those maps N , and we measured the corresponding computational latency of a single output feature map. The plot validates the latency model proposed in equation 4.2. In Figure 4.8(b), we show how the slope of the measured latency curves, shown in figure 4.8(a), varies with the size of the input feature map (N). We employed Ordinary Least Square (OLS) curve fitting to find the value of β_R at 150 MHz. Similar curves were obtained for the CPU implementation i.e., the

latency was shown to also grow linearly, and the CPU latency model exhibits a similar form to the FPGA latency model. Note that the circuit we designed in section 4.3.2, computes a single output feature map at a time. We can duplicate the circuit to produce several output feature maps in parallel. Consequently, if we deploy D copies of the circuit shown in figure 4.3, the Latency of computing S output feature maps would become:

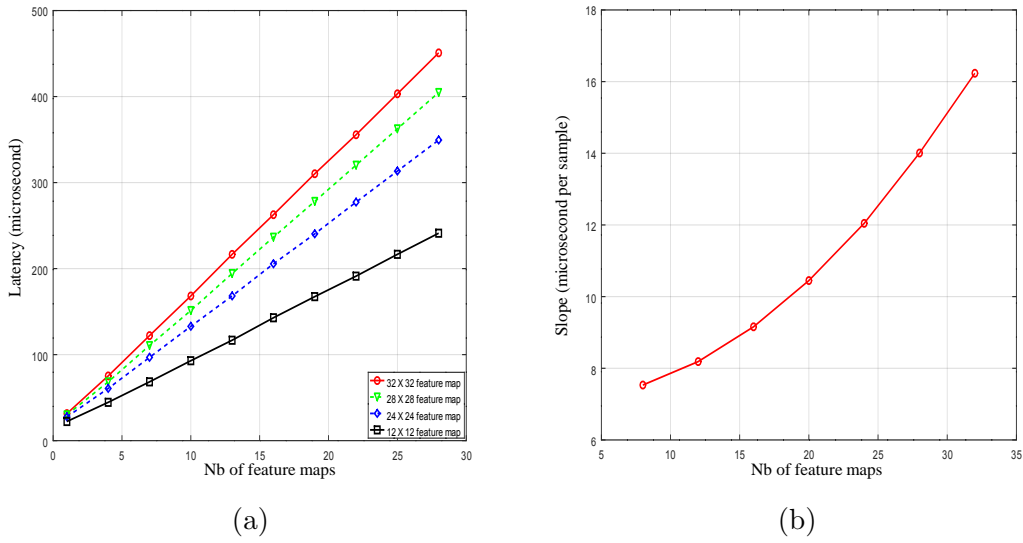


Figure 4.8: Latency of computing a single output feature map on the FPGA and the slope

$$L_{FPGA,5 \times 5} \approx \frac{I \times S \times (N + K - 1)^2}{D \times \beta_R \times f_{clk}} (\mu sec) \quad (4.3)$$

Similarly, the computational latency, in microseconds, when computing S output feature maps on the CPU processor core may be estimated by:

$$L_{CPU,5 \times 5} \approx \frac{I \times S \times (N + K')^2}{\alpha_{R,cpu}} (\mu sec) \quad (4.4)$$

$\alpha_{R,cpu}$, here, is a parameter that characterizes the performance of the CPU

processor core. For large values of I , the *speedup ratio* may be estimated using the following formula:

$$Speedup \approx \frac{L_{CPU,5 \times 5}}{L_{FPGA,5 \times 5}} = \frac{D \times \beta_R \times f_{clk}}{\alpha_{R,cpu}} \times \frac{(N + K')^2}{(N + K - 1)^2} \quad (4.5)$$

In order to identify the maximum possible *speedup ration*, we evaluated the limit of this speedup as N grows arbitrarily. Equation 4.6 reports the maximum achievable speedup.

$$\text{maximum speedup} \approx \frac{D \times \beta_R \times f_{clk}}{\alpha_{R,cpu}} \quad (4.6)$$

In table 4.1, we have shown that the circuit described in figure 4.3, consumes at most 11.36% of the DSP resources. The remaining FPGA resources can be used to deploy 7 additional convolution circuits, where each convolution circuit may compute a different output feature map in parallel. Moreover, $\alpha_{R,cpu}$ for the Core i7 processor core and $\beta_R \times f_{clk}$ were estimated using Ordinary Least Squares and were found to be $5.9859 \times 10^6 \text{ sample/sec}$ and $1.0035 \times 10^8 \text{ sample/sec}$ respectively. Knowing that, in the setting described in section 4.4.1, D can be 8, the maximum achievable *speedup ratio* can be calculated using equation 4.6 as shown below:

$$\text{maximum speedup} \approx \frac{8 \times 1.0035 \times 10^8}{5.9859 \times 10^6} = 134.115 \quad (4.7)$$

4.4.5 Energy Saving Resulting from employing the FPGA-base accelerator

For the energy efficiency test, we measured the power consumption of the Zed-Board with and without FPGA-based acceleration. The measured board power

of the ZedBoard was found to be around 2.8 Watts in both cases. In the previous section, we found that the FPGA implementation of the multi-layer convolution, on the ZedBoard, may even outperform a Core i7 processor core. We also found that the computational latency when computing S output feature maps from I input features may be estimated using equation 4.3, when I is large. The computational latency on a CPU core may be estimated using equation 4.4. A Quad-core Core i7 processor core consumes at least 85 Watts, even when it's idle. Computing S output feature map on the ZedBoard, consumes $E_{ZedBoard}$ joules of energy:

$$E_{ZedBoard} \approx L_{FPGA} \times P_{ZedBoard} \quad (4.8)$$

Similarly, computing S output feature maps on a Core i7 processor core, consumes $E_{Core\ i7}$ joules of energy:

$$E_{Core\ i7} \approx L_{CPU} \times P_{core} \quad (4.9)$$

If we define the Energy Reduction Ratio (ERR) as the ratio of the energy consumed by the processor core to the energy consumed by the FPGA compute node, then:

$$ERR \approx \frac{E_{core}}{E_{ZedBoard}} = \frac{P_{core}}{P_{ZedBoard}} \times \frac{D \times \beta_R \times f_{clk}}{\alpha_{R,cpu}} \times \frac{(N + K')^2}{(N + K - 1)^2} \quad (4.10)$$

4.5 Conclusion

In this work, we aim at deploying FPGA-based compute nodes in the data center environment. This was achieved by devising an FPGA-based accelerator system design for distributed task-loads in a data-center-like environment, and by extending the Spark cluster computing environment to let it target the accelerators. Experimentations on a single FPGA-based compute node with an accelerator circuit set to perform multi-layer convolution operations, one of the most computationally expensive operations during both inference and training of Convolutional Neural Networks, have shown that FPGA-based acceleration, on a single node, may provide orders of magnitude speedup over software implementations on a Core i7 processor core. In addition, we have shown that employing an FPGA-based accelerator system in the data center may significantly reduce power and energy consumption. To summarize, FPGA-based acceleration may provide a very good compromise between customization and flexibility; Moreover, the good performance per energy figures FPGA platforms may provide can be harnessed to efficiently scale up the performance of data center facilities.

Chapter 5

FeatherNet: An Accelerated Convolutional Neural Network Design for Resource-Constrained FPGAs

With the proliferation of IoT- based applications, a surge of interest in embedded vision systems has emerged in both the industrial and research communities. Applications of embedded-based vision systems include, among many others, Unmanned Aerial Vehicles (UAVs) with object/face detection/localization capabilities, pedestrian detection systems in autonomous vehicles, and smart domestic robots. In the data center environment, vision systems may also be employed on a massive scale. For instance, in the future, search engines, with reverse image search capabilities, may employ computer vision components to identify images with a desired content [9–11]. On the other hand, recent advancements in machine learning have led to a widespread rise of what is commonly known as deep learning applications. Deep Convolutional Neural Networks, also known as ConNets or CNNs, are a special kind of deep machine learning models inspired by the structure of the mammalian visual cortex [12]. Nowadays, the state-of-the-art in computer vision systems employs deep ConNets [13, 30, 32]. However, these

networks are very compute intensive in both the training and inference phases, and their success in recent years has been mainly due to two factors: (1) the unprecedented computing power made available by general purpose computing platforms and (2) the availability of large collections of labeled training datasets.

Typically, IoT and embedded vision systems are limited in both computational processing and power capacity [101,102]; moreover, these platforms should be available at the lowest possible cost to economically justify their use [103]. Consequently, deploying pre-trained CNNs on embedded vision platforms to harness their inference ability must rely on energy efficient custom-designed accelerators that are adapted for convolutional neural computations. Among acceleration technologies, Application Specific Integrated Circuits (ASICs) provide the best performance per energy figures as they can be strictly tailored for a certain application. However, they lack the flexibility of general-purpose computing platforms such as CPUs and GPUs, and thus they fail at adapting to fast changing deep learning algorithms. Field Programmable Gate Arrays (FPGAs), on the other hand, provide a very good balance between hardware acceleration, customizability, and flexibility, through reconfigurability. A lot of work in the research literature targets high-end FPGA platforms and can achieve very high computational throughput (e.g. 1020 img/sec [26]).

In this work, we present an efficient FPGA hardware template architecture for deep neural inference tailored for low-end, resource-constrained FPGA platforms, which are commonly aimed at edge computing and IoT applications [33–36], but that are increasingly being used for high-end, computationally and memory intensive CNN applications [13, 30–32]. This chapter advances knowledge by providing a new design methodology that achieves the least amount of resources and that targets low-end FPGA platforms for IoT deployment. Additionally,

as part of the design process, new methods and techniques were introduced to address several design challenges. We believe that the solutions to these design challenges can benefit designers and other researchers using similar devices or facing similar challenges:

1. *For efficient signal processing with reduced resource utilization*, we present a novel stride-aware graph-based method targeted at ConvNets. The method is inspired by previous literature, in particular the first Noble identity from the multi-rate DSP literature [24, 104]. In our work, we differ from the previous DSP literature in that we modified the standard Noble identity to a new form (shown in figure 5.7(b) and proved in appendix A) that can be employed to obtain resource-efficient implementations of 2D-convolutions with strides. This modified form of the Noble identity can be repeatedly employed in a series of transformations on signal-flow-graphs for 2D-convolutions to derive compact and minimal convolution filter architectures that can be directly implemented in hardware. In addition to the convolution filters with strides, we also modeled the other computations involved in ConvNets (e.g. 2D-convolutions with stride 1, pooling layers, and normalization layers), and derived corresponding resource-efficient implementations.
2. *For determining the minimal precision arithmetic needed while preserving high accuracy*: we propose variable-width dynamic fixed-point representations combined with a layer-by-layer design-space pruning heuristic across the different layers of the deep ConvNet model. This is different from the literature where either a single fixed-point representation is chosen at all layers, or the search space is exhaustive and slow. The selected word lengths

are also constrained to being multiples of the model parameter word lengths to simplify data storage and movement across the model.

3. *For achieving a modular design that can support different types of ConvNet layers while ensuring efficient resource utilization*, we propose the computational modules to be relatively small. In the FeatherNet design, the modules are composed of computational filters that can be interconnected to build an entire accelerator design. These model elements can be easily configured through HDL parameters (e.g. layer type, mask size, stride, etc.) to meet the needs of specific ConvNet implementations and thus they can be reused to implement a wide variety of ConvNet architectures. Although we used these model elements to implement AlexNet [13], the modular nature of the design allows the easy implementation of other ConvNets as well. In the context of targeting resource-restricted FPGAs, the modular nature of our design allows us to easily remove elements from an overall design without affecting the flow of data within it, and thus save FPGA resources. In this aspect, our approach is different from previous modular ConvNet designs, where the modules are composed of relatively large blocks of logic resources in contrast to smaller blocks of filters, as in our case.

4. *For ease of portability between two different FPGA vendor platforms, namely Intel/Altera and Xilinx*, we make sure that our HDL implementation is not specific to one FPGA product, and that the design is portable across two competing FPGA vendor platforms. For proof of concept, we proved this principle on Intel/Altera and on Xilinx. Previous approaches [105, 106] achieve similar portability by implementing their designs in behavioral VHDL/Verilog, and rely on the synthesis tools to resolve the micro-

architectural differences and infer the device-specific components for each target device (BRAMs, DSPs, etc.). The limitation of these approaches is that the synthesis tools do not have information about the purpose or intent of the design, and thus may try to optimize for area and performance by inferring device primitives. As a result, the tools may misplace the individual hardware blocks (e.g. BRAMs and LUTs) in certain sub-components of the design. To address this limitation, we rely on instantiating the device-specific hardware blocks needed in each computational filter rather than simply relying on the tools to infer these blocks while keeping track of the similarity and differences between the platforms. For example, due to the dimensions of the ConvNet feature maps involved in the computations in AlexNet, normalization filters are best implemented using BRAMs, whereas max pooling filters are best implemented using LUTRAMs in Xilinx FPGAs [33] and MLABs in Cyclone V [34]. To achieve the desired portability, we develop a hierarchical design that decouples device-specific features needed to meet the low resource utilization target from the rest of the design. We use VHDL generics to seamlessly instantiate and integrate the device-specific components in our design. Using this decoupling, we are able to successfully implement our ConvNet on two architecturally different FPGA devices; the Xilinx Zynq and Intel Cyclone V, using two different logic synthesis tools. Our design can be easily ported to other devices by editing a small number of VHDL source files.

Our methodology is applied to AlexNet [13], a popular CNN for image classification that requires 700 million multiplications with 61 million parameters for each image. Although the implementations were customized for the AlexNet CNN, the architecture can be adapted for other convolutional network architec-

tures. Our results demonstrated the success of addressing the design challenges and achieving low (30%) resource utilization for the lower end FPGA platforms: Zedboard [35] and Cyclone V [36]. The design overcame the limitation of designs targeted for high end platforms which cannot fit on low end IoT platforms [101–103]. Furthermore, our design showed superior performance results (measured in terms of performance/watt/dollar) compared to high end optimized designs (9.31×10^{-3} Frame per Joule per Dollar compared to 5.17×10^{-3} for the state-of-the-art [26]). The designs also showed accurate results consistent with non-accelerated designs, where images from the ILSVRC-2012 dataset were classified with a top-5 accuracy of 83.58% (compared to 79% in [26]) while achieving the low-end FPGA benefits of improved energy efficiency per cost at 9.31×10^{-3} Frame per Joule per Dollar and a frame rate of up to 9 *frames/sec*.

The rest of this paper is structured as follows: Section 5.1 provides a detailed background on Convolutional Neural Networks. We also shed some light on a selection of the latest related research efforts on accelerating CNNs on FPGAs, in section 5.2. Section 5.3 illustrates the design methodology used to realize efficient implementations of the different computational sub-tasks involved in ConvNets. In Section 5.4, we describe the architecture of our proposed accelerator. Finally, we evaluate the performance and energy efficiency of the design in Section 5.5.

5.1 Deep Convolutional Neural Networks

Deep Convolutional Neural Networks (ConvNets) are a special kind of feed-forward multi-layer neural networks (NN) that typically consist of a series of convolutional and pooling layers, followed by one or more fully-connected layers. In a convolutional layer, the neurons are arranged into several separate rectan-

gular patches commonly known as Feature maps [12]. This arrangement makes them suitable for processing multi-dimensional data such as 2-D color images and 3D video signals. In typical multi-dimensional signals such as images and videos, local groups of pixels are highly correlated [12]. Convolutional layers are designed to exploit this spatial correlation in an input neural layer by employing a local connectivity pattern between neurons in consecutive layers; in other words, in a convolutional layer, every neuron is connected to only small patches of neurons, commonly referred to as local receptive fields, residing on different feature maps from the previous convolutional layer. Another key aspect that distinguishes convolutional layers from fully connected NN layers is the parameter sharing scheme in which all neurons that belong to the same convolutional layer share the same set of weights. These arrangements, i.e., local connectivity and parameter sharing, are mathematically described by a multi-dimensional discrete convolution operation followed by a non-linear activation function [12, 25]. Figure 5.1 illustrates the structure of a typical convolutional layer. A convolutional layer can assume multiple hyper-parameters. Among these is the size of the local receptive field, which is the common size of the convolution masks in the 2-D convolutions shown in Figure 5.1. In practical ConvNets, the convolution mask can stride multiple pixels at a time, and thus the stride configuration is another hyper-parameter of a convolutional layer. Another important hyper-parameter is the zero-padding configuration which determines the number of zero samples concatenated at the beginning and at the end of every feature map.

The last hyper-parameter for a convolutional layer is the group parameter, which determines the number of groups the input and output feature maps are divided to. Every group of output feature maps is then connected to its corresponding group of input feature maps. This arrangement reduces the complexity

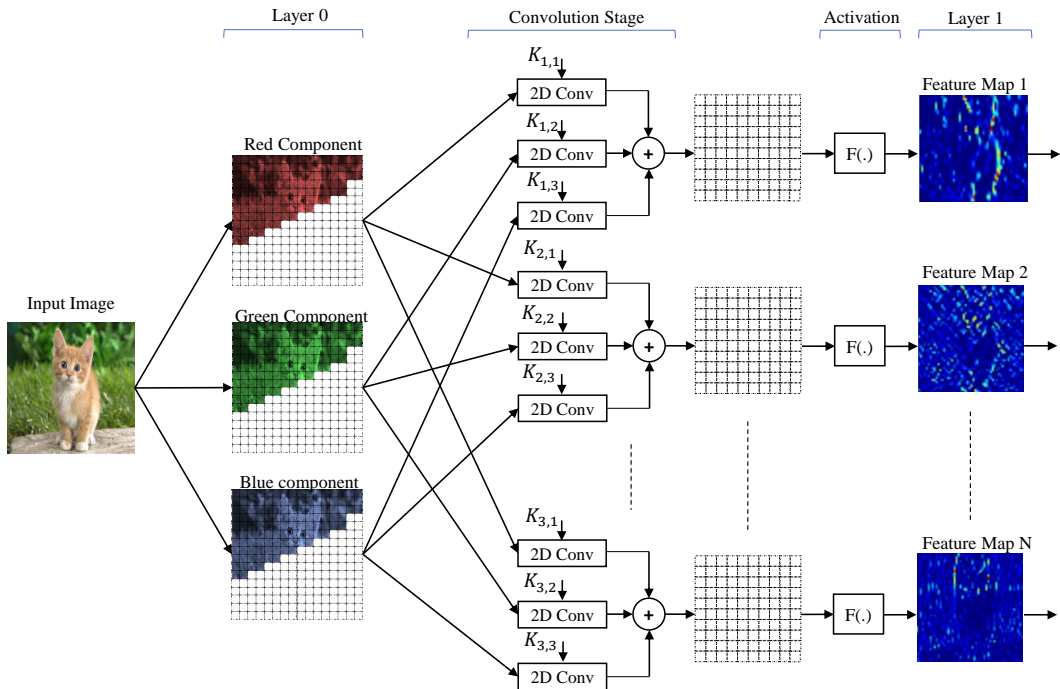


Figure 5.1: An illustration of a typical Convolutional Layer.

of the ConvNet, but it is rarely used nowadays since the current usually powerful computing platforms obviate the need for it. A convolutional layer is often followed by a pooling layer, which computes a statistical summary of local neural activities in a convolutional layer. Feature maps at the input of the pooling layer are first divided into rectangular patches of neurons, also referred to as receptive fields. The neurons in every patch or receptive field are then summarized into a single neuron for every patch, typically using the maximum of the samples or the average of the samples. The purpose of pooling is to make the detection invariant to minor shifts and distortions [12].

Currently, deep ConvNets employ more processing stages between the convolutional and pooling layers. For example, AlexNet [13], a ConvNet used for image classification, employs a Local Response Normalization (LRN) layer after the first and second convolutional layers. The role of the LRN layer is to drain

Table 5.1: AlexNet layers and their hyper parameters [13].

Layer Name	Type	Hyper Parameters			
Conv1	Convolutional	Size: 11×11	Stride: 4	Pad: 0	Group: 1
Norm1	LRN	$\alpha/k = 5 \times 10^{-5}$	$\beta = 0.75$	n: 5	
Pool1	Max Pool	Size: 3×3	Stride: 2	Pad: 0	
Conv2	Convolutional	Size: 5×5	Stride: 1	Pad: 2	Group: 2
Norm2	LRN	$\alpha/k = 5 \times 10^{-5}$	$\beta = 0.75$	n: 5	
Pool2	Max Pool	Size: 3×3	Stride: 2	Pad: 0	
Conv3	Convolutional	Size: 3×3	Stride: 1	Pad: 1	Group: 2
Conv4	Convolutional	Size: 3×3	Stride: 1	Pad: 1	Group: 2
Conv5	Convolutional	Size: 3×3	Stride: 1	Pad: 1	Group: 1
Pool5	Max Pool	Size: 3×3	Stride: 2	Pad: 0	
Fc6	Fully Connected	Input Neurons: 9216		Output Neurons: 4096	
Fc7	Fully Connected	Input Neurons: 4096		Output Neurons: 4096	
Fc8	Fully Connected	Input Neurons: 4096		Output Neurons: 1000	

the response of local clusters of neurons that have uniformly large activations and boost those neurons that have relatively large activations when compared to their neighbors. In the last few years, many other ConvNet architectures were inspired by AlexNet and were able to achieve better performance. Those include GoogleNet [30], Zeiler& Fergus [31], and VGG [32]. AlexNet, however, is still widely used as benchmark for evaluating ConvNet accelerators [26, 107] due to its high computational complexity. Table 5.1 summaries the AlexNet computational layers and the hyper-parameters we used in our implementation.

5.2 Related Work

The work of Farabet et al. [27, 89, 90], was among the first pioneering efforts to accelerate Convolutional Neural inference on low-end DSP-oriented FPGA platforms. In [89], a programmable FPGA-based ConvNet Processor (CNP) was designed and deployed on two different FPGA platforms: a Xilinx Spartan-3A DSP 3400 FPGA and a Xilinx Virtex 4 SX35 FPGA. The CNP consists of a Parallel Vector Arithmetic and Logic Unit (VALU) along with a control unit

that sequences the operations of the VALU. The CNP could process 10 frames per second using a ConvNet with 2 million synaptic connections. Current generations of ConvNets, however, are much more complex; AlexNet, for instance, has around 724 million synaptic connections. Moreover, the ConvNets that were deployed in [27, 89, 90] do not contain many of the more intricate structures and layers involved in current state-of-art ConvNets. This means that even though the works in [27, 89, 90] target low-end platforms, they do not support recent ConvNet architectures such as AlexNet [13], GoogleNet [30], and VGG [32].

In general, any convolutional layer in a ConvNet inference workload can be essentially resolved into multiple multidimensional Finite Impulse Response (FIR) convolution operations [8, 12]. Implementing N-dimensional FIR Convolutions on FPGA platforms was investigated in [108–112]. The work of Meher et al. [108] explores the realization of 1D- and 2D- FIR filters using systolic decomposition of Distributed Arithmetic (DA) based inner-product computations. The resulting DA-based FIR implementations can be implemented using Lookup tables and shift-add operations only, and thus do not require any multipliers. While these implementations can be efficiently mapped to LUT-based and legacy FPGA devices, they provide little benefit on existing and modern DSP-based FPGA devices, i.e., FPGA devices that incorporate special reconfigurable hardware multipliers referred to as DSP units. Moreover, implementing FIR filters using distributed arithmetic assumes that the coefficients of the FIR convolution filters are constant. Since modern ConvNet architectures require thousands of convolution operations per single inference, the best implementation strategy for large ConvNet architectures relies on implementing multidimensional FIR convolution filter implementation with variable convolution coefficients. Several other notable optimizations were proposed in the literature: the use of Partial Buffering (PB)

scheme to implement 2D- Convolutions on FPGA devices [109, 110], the use of multi-window PB to improve tradeoff between resource utilization and external memory bus bandwidth [111], and the combination of Carry-save and Carry-propagate arithmetic to implement convolution adder trees on FPGA devices that supports Carry-propagate arithmetic only [112]. In this work, we address the implementation of convolution filter architectures for 2D- convolutions with arbitrary strides and mask sizes tailored for implementing complex ConvNet architectures on resource-constrained FPGA devices. We describe the proposed design methodology in section 5.3.

In recent years, interest in accelerating neural network inference on FPGA platforms has grown significantly and different design approaches were investigated and employed. In the work of Zhang et al. [25], a design space exploration methodology was proposed. Various computation optimizations and transformations such as loop unrolling, pipelining, and tiling were investigated along with memory access optimization schemes. With the help of the roofline performance model, proposed in [113], optimized solutions were identified, which were then implemented using High Level Synthesis tools [114, 115]. The work of Zhang et al, [25] inspired a series of other research efforts such as that of Alwani et al. [116], which observed the existence of a previously unexplored design dimension that focuses on fusing the processing of multiple convolutional layers, and that of Shen et al. [117] which proposed an automated design methodology that divides the FPGA resources into several processors instead of a single large processor achieving more throughput.

The automatic generation of FPGA accelerators from a high-level description of a Convolutional Neural Network was also investigated and implemented by Sharma et al. [107] who proposed a framework, called DNN-Weaver, that auto-

matically generates a synthesizable hardware design for Deep ConvNets. DNN-Weaver was used to generate designs for three high-end FPGAs: the Intel Arria 10 and Stratix V, and the Xilinx Zynq FPGA.

The current state-of-the-art appeared in the work of Aydonat et al. [26], who proposed a novel architecture written in OpenCL [115] called Deep Learning Accelerator (DLA). The DLA employs several design techniques such as maximizing data reuse and external memory bandwidth and the use of the Winograd transform to boost the performance of the implementation. The DLA could process 1020 images per second using AlexNet deployed on a high-end Arria 10 FPGA device.

HDL-based implementations of ConvNet architectures were also explored in the literature [105, 106]. In [105], an open source VHDL-based ConvNet library and toolbox is presented. The toolbox provides an easy method to investigate the implication of employing low-bit fixed-point arithmetic on each individual layer, while the VHDL library provides reference designs for different ConvNet layers (e.g. Hardware synthesizable neurons and a PCIe-based ZF-net [31] layer implementation). The final purpose of this library is to deploy ConvNet inference algorithms on high-end Xilinx Ultrascale FPGA platforms tailored for data center deployment. In the work of Abdelouahab et al. [106], a framework called Haddoc2 was proposed. The Haddoc2 framework can take a Caffe model for a very small ConvNet architecture and automatically generate an equivalent VHDL-based hardware description of the network. The framework also employs other optimizations such as using short fixed-point arithmetic and implementing multiplications using Logic Elements (LE) rather than hardware DSP blocks. As in [105] and citeref:Abdelouahab, in this work, we resort to implementing our accelerator in an RTL language, in particular we employ a combination of

VHDL and Verilog. This choice of design entry language was made to avoid the limitations of High Level Synthesis tools described in reference [118].

Implementing deep ConvNet inference in reduced precision fixed-point arithmetic has also been investigated in the literature and the existing approaches generally fall into two categories. The first category, also known as the post-training quantization approach, relies on training deep ConvNets in floating-point and then performing a floating-to-fixed-point conversion where for a given ConvNet layer the output neural activations and the neuron parameters are both represented in fixed-point [119–122]. The second category relies on trained quantization, where the network is trained with the constraint of having quantized weights [123–129]. Notable efforts that employ this approach include training Binarized Neural Networks (BNN) whose weights and neural activations are constrained to +1 or -1 [123, 128], or training Ternary Weight Neural Networks (TWN) whose weights are constrained to +1, 0, and -1 [127], or training ConvNets that have reduced precision weights and activations [124–126, 129]. In the work of Jacob et al. [126] a quantized training framework was proposed to minimize the loss of accuracy from quantization, and the experimental results have shown significant improvements in the tradeoff between accuracy and on-device latency.

In this work, we focus on the development of accelerator designs that enable the implementation of complex ConvNet architectures on resource-constrained FPGA platforms aimed at the IoT market, and thus we seek implementations that consume the least resources possible to target low-end FPGA devices while still achieving acceptable performance. Our work contrasts with the majority of FPGA-based ConvNet accelerator designs reported in the literature, which either leverage the abundant resources of mid- to high-end devices to achieve

the highest performance possible when implementing complex ConvNets [25, 26, 105, 107, 116, 117], or resort to implementing relatively small ConvNet architectures that have limited storage and computational requirements on embedded systems [106, 119, 126]. To tackle the challenge of designing a resource-efficient ConvNet accelerator, we devise a two-fold methodology. First, we propose a set of data-flow-based design techniques that allow a designer to extract DSP-based graphical representations of deep ConvNet sub-computations and map these representations to efficient hardware implementations. In this context, we also propose a novel stride-aware graph-based method targeted at convolutional layers that employ 2D-convolutions with non-unit strides; this method can be systematically employed to derive compact and minimal-resource 2D-convolution filter architectures that can be directly implemented in hardware. Second, we investigate employing dynamic fixed-point arithmetic, i.e., different layers of the ConvNet use different bit-widths. Our approach is similar to the approaches used in references [119] and [122]; however, we differ from these approaches in that we use a layer-by-layer design-space pruning heuristic to infer the minimal required bit-widths and precision-levels in each ConvNet layer.

5.3 Design Methodology

5.3.1 Design Challenges

For the design to meet the performance, power and cost requirements of IoT applications, we identified four challenges in designing the accelerator:

1. The limited resources available on low-end FPGA platforms. These platforms are well suited for low-cost and power sensitive applications and are

typically characterized by a limited number of available DSP units, Block RAMs and programmable logic blocks. They are usually promoted by their manufacturers as capable of delivering the highest DSP performance-per-watt.

2. The relatively high computational complexity of deep convolutional neural networks. The number of multipliers in an AlexNet [13] inference is on the order of 700 million operations per frame. Given the real-time requirement of IoT applications, which is several frames per second and requires billions of operations per second, this places a significant strain on the resource-limited FPGA.
3. The memory or space requirements of deep convolutional neural networks, where each convolutional layer can produce many feature maps. The following convolutional layer may use these feature maps multiple times, which requires caching the intermediate feature maps.
4. The limited on-board memory bandwidth in low-end platforms. The memory bandwidth on such devices is typically limited to around 100 Gbps. The Cyclone V dev-kit, for instance, has a theoretical memory bandwidth of 70 Gbps and an achievable rate of 58.9 Gbps. The Zedboard, another low-end platform, can achieve around 34 Gbps. These numbers can be contrasted with those of high-end platforms such as the Intel Arria 10 and Stratix V that have 273 Gbps and 372 Gbps respectively.

To deal with these challenges, we propose several design techniques for implementing convolutional neural networks on resource-constrained FPGAs. The first design technique is based on modeling sub-computations in convolutional

neural networks in terms of graphical representations, and then employing a set of high-level transformations to derive suitable resource-efficient realizations. The second design technique is based on implementing the accelerator using limited precision fixed-point arithmetic, which greatly reduces resource utilization on the FPGA.

5.3.2 Graphical Representation and Modeling of Neural Inference computation

The computations involved in convolutional neural network inference are best described in terms of Digital Signal Processing (DSP) computations or signal processing chains rather than general purpose workloads. DSP computations differ from their general-purpose counterparts in two major ways. The first difference is in the real-time requirement where data is received periodically from a source and should be processed within a bounded time frame, as in a real-time vision system with a camera generating frames at a certain rate. The second difference is in the data-flow property where a signal processing sub-task starts executing as soon as the data needed for that sub-task is available at the input. This modeling of computations is usually referred to as the streaming data-flow compute model, in which the computations can be modeled as a set of transformations on an input data-sequence rather than a sequence of operations that needs to be performed on a batch of data. The study of modeling DSP computations and data-flow algorithms is very well established in the literature, especially, in the work of Keshab [104], and later Hauck and Dehon [24]. In this work, we reason about the implementation of various sub-tasks involved in convolutional neural networks using the data-flow compute model, in which computations are

described using a Graphical representation, that is, a Data-Flow Graph or a Block Diagram. We, then, propose a set of high-level transformations on these graphical representations to derive efficient hardware implementations for many computations involved in deep convolutional neural networks.

Efficient Modeling of 2-D Convolutions.

The most important, and computationally intensive sub-task in a convolutional neural network, is the multi-layer convolution operation, which consists of a sequence of 2-dimensional discrete-time convolution operations. Every convolution operation involved can be modeled as a discrete linear shift-invariant system with a finite impulse response; consequently, every convolution is, in principle, a Finite Impulse Response (FIR) filter. In this section, we establish a design methodology for deriving hardware implementations for 2-D FIR convolution filters by extending the previously known methods in the digital signal processing literature. Figure 5.2(a), shows a 1-dimensional 3-tap FIR filter in block diagram representation. For the sake of simplicity and presentation, we will use the 3-tap FIR filter as a typical example to illustrate the proposed design techniques. Note, that the block diagram representation of the filter clearly exposes the data-flow properties of the 1-D convolution, including its data-driven properties, and it unmask the inherent fine-grain parallelism among different operations in the convolution. The representation shown in Figure 5.2(a), can be mapped directly to a hardware implementation that consists of three independent hardware multipliers, two adders, and two delay elements which can be realized using two hardware registers. The filter can convolve an input sequence with an impulse response sequence of size three. At every iteration or clock cycle, the filter reads one input sample from the input sequence, performs three multiplications and 2

additions in parallel and generates one output sample. Another architecture can be derived for the block diagram representation; one that can perform the same convolution with less hardware multipliers and adders, but such an implementation is more serial in nature and cannot process a sample on every iteration. In this work, we seek implementations that can at least process one input sample on every iteration.

Although designing a 2-D convolution filter is slightly more complex, it can be derived from its 1-D counterpart by exploiting the fact that a 2-D convolution is essentially made of several concurrent 1-D convolution operations. A high-level mathematical formulation of the 2-D convolution operation is illustrated in equation 5.1.

$$I' = \sum_q \sum_p K(p, q) \times I(x - p, y - q) \quad (5.1)$$

$I(x, y)$ is the input 2-D sequence or image. $K(x, y)$ is a 2-D sequence that represents the convolution kernel or the filter mask. $I'(x, y)$ is the 2-D sequence that results from convolving $I(x, y)$ with $K(x, y)$. It may be shown that equation 5.1 can be reformulated in terms of simple 1-D convolutions. The reformulation is shown in equation 5.2

$$I'_x(y) = \sum_q K_x(q) * I_x(y - q) \quad (5.2)$$

$k_x(y)$ denotes the y^{th} row of the filter mask; $I_x(y)$ also denotes the y^{th} line or row of the input image, and the “*” binary operator denotes the 1-D linear convolution. Note that although equation 5.2 describes a 2-D convolution operation, it exhibits a great deal of similarity to the that of a 1-D convolution, but with the multiplication operator being replaced by a 1-D convolution operation

denoted by “*”. This similarity allows us to draw a comparable block diagram representation for the 2D-FIR filter as shown in Figure 5.2(b). The 2 representations shown in both Figure 5.2(a) and 5.2(b) have a great degree of similarity in their structures, with the only difference being in the operations used.

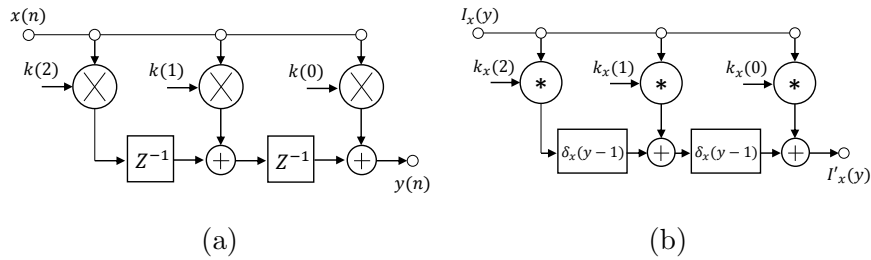


Figure 5.2: Block diagram representation for the 1-D FIR filter (a) and for the 2-D FIR (b)

The block-diagram representations shown in Figure 5.2 are different in two respects:

- In the first diagram, the filter consumes 1 sample from the input sequence at every iteration; similarly, every operation in Figure 5.2(a) processes one sample per iteration. The diagram shown in Figure 5.2(b) receives an entire row of samples from the input 2-D sequence at every iteration and every sub-task in this diagram operates on an entire row of samples rather than on a single sample.
- In figure 5.2(b) the delay elements denoted by Z^{-1} are replaced by multi-dimensional (2D) linear systems [130] with an impulse response of $\delta_x(y-1)$ each, which is a fancy way to refer to delays along the y-axis. In other words, if we apply a 2-D sequence $S(x, y)$ to the input of this system, the output will be the sequence $R(x, y) = S(x, y-1)$.

Many hardware realizations for the 2-D FIR filter can be derived from Figure

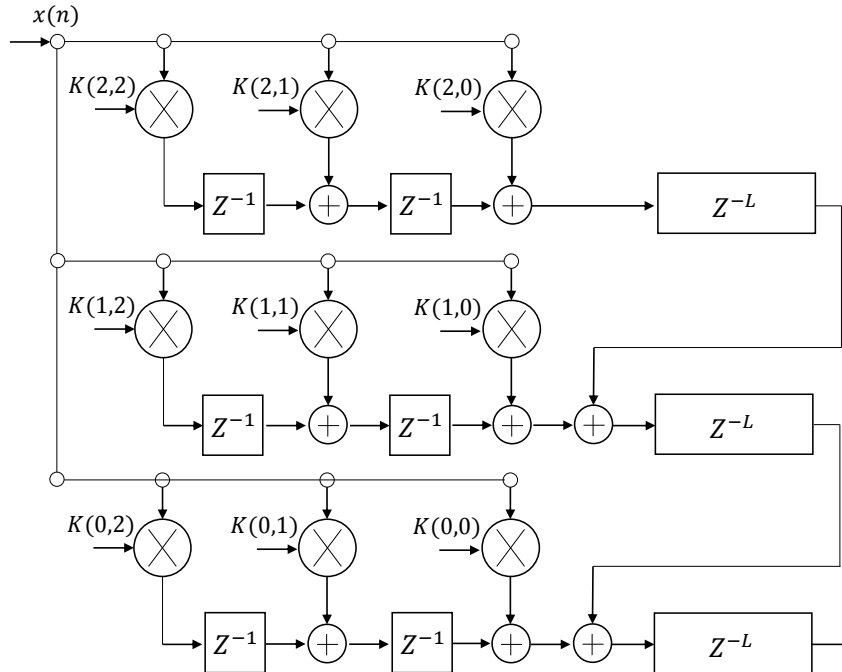


Figure 5.3: A hardware realization for the 2-D convolution filter derive from figure 5.2(b).

5.2(b). However, we can set for a minimal-resource realization, that is a realization that takes a minimal number of multipliers and adders and can process at least one input sample on every iteration. We show this realization in Figure 5.3. With this realization, the 2-D input sequence is processed one row at a time. Equally, every row is processed one sample at a time. This means the 2D input sequence should be received by the filter in row major order; and the resulting output sequence is also generated in row-major order. As a result, in this work, all input feature maps are stored and transferred in this particular order.

Different convolution filters with different filter mask sizes, can be designed using the same methodology we used to design the 2-D FIR filter with a mask size of 3×3 . In general, a hardware realization of a 2-D convolution filter with a filter mask of size $N \times M$ and designed according to our methodology would result in

Table 5.2: Hardware cost of different 2-D FIR filters

FIR window size	Multipliers	Adders	Registers
3x3	9	8	6
5x5	25	24	20
7x7	49	48	42
11x11	121	120	110

$N \times M$ hardware multipliers, $N \times M - 1$ adders, $(N - 1) \times M$ registers, and $M - 1$ delay lines of depth L each, where L is the width of the resulting output feature map. Although, the 3×3 2-D convolution was portrayed in this section as a toy example to illustrate the methodology, this convolution kernel is heavily used, along with other convolution mask sizes, in current deep convolutional neural networks. The following table summarizes the hardware cost figures for many convolution filter implementations.

As shown in Table 5.2, the complexity of filters with large FIR windows such as the 7x7 or 11x11 is relatively high, especially, when working with resource-limited FPGA platforms. The Zedboard for instance has around 240 DSP48E units, implementing a filter with a convolution kernel of size 11x11 would not fit into the Zedboard’s FPGA fabric, if every multiplier is implemented using two DSP48E units. Fortunately, however, in modern convolutional neural networks, some convolutional layers have a stride configuration that differs from 1, i.e., the convolution mask can jump multiple pixels at a time as it slides around the image. For instance, the convolution kernel of size 11x11 usually has a stride configuration of at least 4 pixels and hence an 11x11 FIR with a stride configuration of 4 may be implemented with only 9 multipliers instead of 121, greatly reducing its cost. In section 5.3.3, we propose a set of high-level transformations and reduction schemes to design hardware implementations for FIR filters with

arbitrary stride configurations.

Modeling the Local Response Normalization Layer

The concept of Local Response Normalization is borrowed from the phenomenon of “Lateral inhibition” in neurobiology [12]. In biological neural networks, a sturdily excited neuron has the capacity to quell the neighboring neurons improving the perception ability of the entire network. The first effort to introduce the concept of Normalization to improve the accuracy of artificial deep convolutional neural networks was in the work of Alex Krizhevsky et al. [13]. The operation for the Local Response Normalization as it appears in [13] is described in equations (3) and (4):

$$b_{x,y}^i = a_{x,y}^i \times c_{x,y}^i \quad (5.3)$$

$$c_{x,y}^i = \left(1 + \frac{\alpha}{k} \sum_{j=i-n/2}^{j=i+n/2} (a_{x,y}^j)^2 \right)^{-\beta} \quad (5.4)$$

Where $a_{x,y}^i$ denote the activity of a neuron that belongs to the i^{th} feature map at position (x,y) and $b_{x,y}^i$ represent the normalized neural activities. Note that the sum in $c_{x,y}^i$ runs over multiple adjacent feature maps at the same position. Hence, the 2-D arrays $c_{x,y}^i$ depend on n adjacent feature maps. We treat the normalization operation as a non-linear DSP filter and thus it can be represented in a Data-Flow Graph representation or in a block diagram representation as shown in figure 5.4. Figure 5.4 illustrates a normalization filter with $n = 3$ feature maps.

$f(x) = (1 + \alpha/k \times x)^{-\beta}$ is a non-linear function; and the $\delta_{x,y}(i-1)$ block is a delay unit along the i-axis. The above block diagram can be directly mapped to a hardware implementation, with two multipliers, two adders and three delay

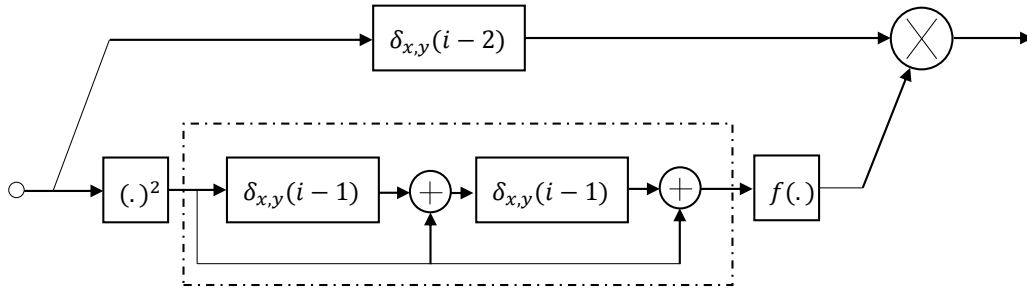


Figure 5.4: A block diagram representation for a Local response normalization filter with $n = 3$.

lines. The non-linear function can be implemented using lookup tables. The lengths of the delay lines depend on the size of the feature maps involved in the normalization layer.

Modeling the Pooling Layer

In traditional convolutional neural networks, a pooling layer operates on non-overlapping rectangular clusters of neurons within the same feature map. In each cluster, the pooling operation condenses the excitations of neighboring neurons into a single neuron in the next layer. In some cases, such as [13], the clusters upon which the pooling layer acts, can be overlapped, thus improving the generalization capacity and accuracy of the trained model. By overlapping these clusters, the pooling layer starts resembling the convolutional layer in that the rectangular clusters of neurons are similar to convolution filter masks. Hence, just like convolutions, the pooling can have a stride configuration that determines how far the pooling clusters are overlapping. In this section, we use a design methodology similar to the one we used for 2-D convolution filters. Although there are many pooling methods, the most common pooling schemes are the max pooling and the average pooling. In max pooling, the maximum excitation value in each

cluster is used, whereas, in average pooling the average of the excitation values is used. A mathematical formulation of a max pooling operation with a stride of 1 pixel (maximum overlap) in both x and y directions, and a mask size of $N \times M$ is shown in equation 5.5.

$$I' = \bigvee_{i=0}^{N-1} \bigvee_{j=0}^{M-1} I(x-i, y-j) \quad (5.5)$$

The \vee operation denotes the maximum operation; mathematically, $\vee(a, b) = \max(a, b)$. Finally, $I(x, y)$ denotes the input feature map and $I'(x, y)$ is the pooled output feature map. Equation 5.5 can be re-written as:

$$I'(x, y) = \bigvee_{j=0}^{M-1} l_x(j) \quad (5.6)$$

In this reformulation, $l_x(j)$ denotes the result of pooling the j^{th} row or line of the input feature map using a 1-D pooling operation. If the feature map arrives at the input of the pooling filter in row-major order, according to Equation 5.6, the 2-D pooling computation can proceed in two stages. In the first stage, every row is pooled individually. In the second stage, the pooled rows obtained from the first stage are combined to obtain the resulting pooled feature map. Figure 5.5, shows a block diagram representation for a 2-D pooling filter with a kernel size of 3×3 and a stride of 1 pixel.

Note that the maximum operation, shown in figure 5.5, is implemented using FPGA carry-chain and lookup table resources which are abundantly available on most FPGAs, as opposed to DSP units which are available in limited numbers. The unit delay elements are implemented using FPGA slice registers, while the delays along the y-axis are implemented using Block RAMs configured as delay lines. Pooling filters with larger kernel sizes and different stride configurations

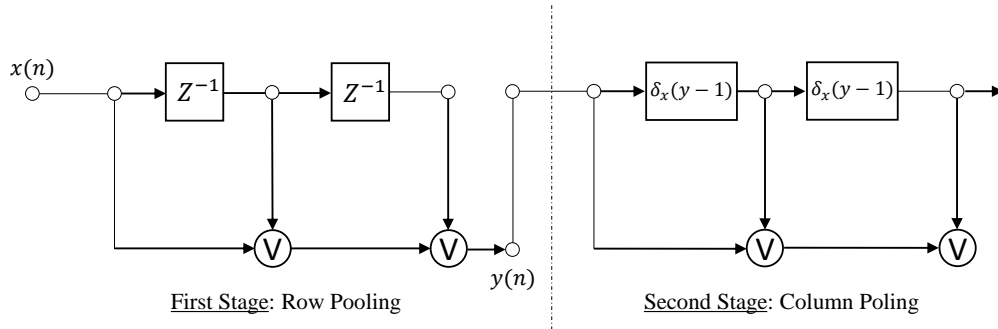


Figure 5.5: A block diagram representation for a 2-D Pooling filter with kernel size 3×3 and a stride of 1 pixel

can be designed in a similar fashion. However, when the stride configuration is different from one, we employ down-samplers at the output of both first and second stages. In the next section, we derive a method to reduce the resource utilization for convolution filters with stride configurations.

5.3.3 Optimization and Reduction schemes for 2D Convolution

As mentioned earlier, in section 5.3.2, some convolutional layers have convolution masks that can jump multiple pixels at a time both vertically and horizontally. We referred to such convolution operations as convolution filters with a stride configuration. We also laid down a systematic technique to derive a hardware implementation for the 2-D convolution filter with a stride of 1 pixel. In this section, we extend our methodology to include implementing convolution filters with arbitrary stride configurations. Designing convolution filters with stride configurations requires a certain understanding of multi-rate signal processing; consequently, we will review some of the notions of multi-rate processing, and then propose a multi-rate-based transformation technique that helps mapping a

2-D convolution with an arbitrary stride configuration to a hardware realization.

Multi-rate Processing Perspective for Strides in ConvNets

In DSP systems, it is possible to reduce the sampling rate of a sequence by an integer factor M by using a sampling rate compressor. The compressor performs periodic sampling of the original sequence at time instances that are multiples of M . To increase the rate, on the other hand, a sampling rate expander is used along with a low-pass filter. In this work, we limit our scope to studying the sampling rate compressor as it can serve our interest in deriving hardware architectures for convolution filters with multiple strides. Figure 5.6, shows the notation of the sample rate compressor and illustrates a typical example of how the compressor reduces the sampling rate of a 1-D sequence. We are going to use this notation to represent our proposed high-level transformation and reduction scheme.

Multi-rate signal processing generally refers to a set of techniques that utilizes sample rate compressors and expanders to reduce the computational cost of some multi-rate signal processing systems and consequently improving their efficiency. Two important results from multi-rate processing are the Noble Identities which allow interchanging of filtering and compressing or expanding operations and the polyphase decomposition of a filter's impulse response. Ref. [131], provides a complete treaty on Noble identities and polyphase decomposition. Figure 5.7(a) illustrates the first Noble identity.

Efficient Multi-rate-based Transformation for Strides

The Noble identity shown in Figure 5.7(a), is used for the analysis of multi-rate systems. M delay elements can be transferred from the input of an M-fold

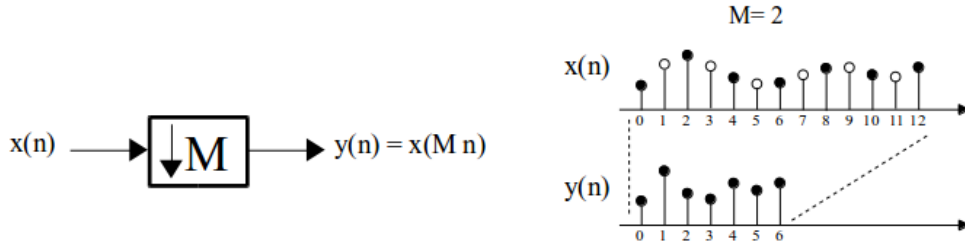
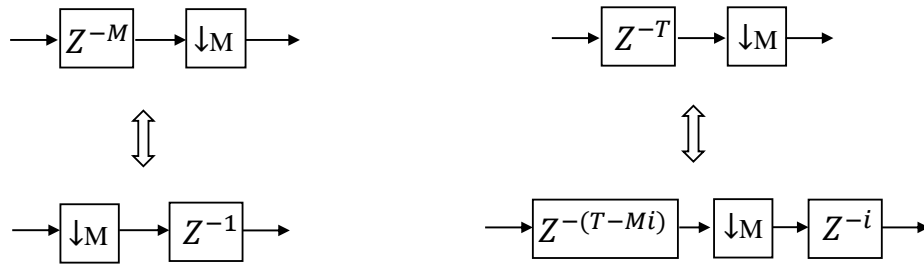


Figure 5.6: A symbolic notation for the compressor and a typical example with $M = 2$



(a) The first Noble Identity

(b) Our proposed transformation

Figure 5.7: Multi-rate signal processing transformations

rate compressor to its output as 1 delay element, as illustrated in Figure 5.7(a). We propose a generalized form of this identity in Figure 5.7(b). Given T delay elements at the input of an M -fold rate compressor, we can transfer $M \times i$ of these elements from the input of the compressor to its output as i delay elements. In this proposed identity, T , M and i can be any combination of natural numbers (we provide a proof for this identity in Appendix A). We will show in the next section, that the identity, shown in Figure 5.7(b), proves to be very useful in deriving hardware architectures for convolution filters with strides.

Designing 2D Convolution Kernels with strides

In this section, we tackle the general case of s_h stride, where the convolution mask can jump s_h samples at a time, where s_h is an arbitrary positive integer greater

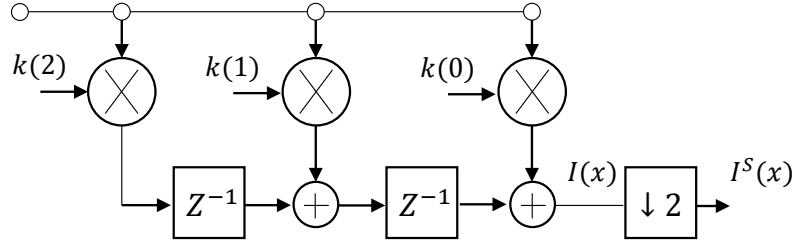


Figure 5.8: A block diagram representation for 1-D convolution with a stride of 2 samples

than or equal to 1. A 1-D convolution with horizontal strides s_h is defined as

$$I^S(x) = \sum_p K(p) \times I(s_h x - p) \quad (5.7)$$

Note that we can break-down equation (5.7) into two steps; the first step consists of performing a normal 1-D convolution with a stride of 1 sample as shown in equation (5.8)

$$I(x) = \sum_p K(p) \times I(x - p) \quad (5.8)$$

The next step consists of down-sampling $I(x)$ to obtain $I^S(x)$

$$I^S(x) = I(s_h \times x) \quad (5.9)$$

Equation 5.11 is an s_h -fold sample rate compressor. By employing an s_h -fold compressor at the output of the normal 2-D convolution, we obtain the same output as that of a convolution with horizontal strides of s_h . Knowing this fact, a block diagram representation for a convolution operation with a stride configuration s_h can be drawn. Figure 5.8 illustrates a block diagram representation for a 1-D convolution filter with a mask size of 3 and a stride of 2 samples.

A direct implementation of the arrangement shown in Figure 5.8 is clearly inefficient, since half of the samples computed by the FIR filter are dropped by the compressor. We employ the transformation shown in Figure 5.7(b), to convert the above diagram to a more efficient data-flow representation in which unneeded computations are avoided. We then fold the data-flow representation to derive an efficient hardware implementation for the 1-D convolution with a stride of 2 samples.

The final adder at the output of the FIR filter (refer to Figure 5.8) and the compressor can be interchanged. The result of interchanging the order of the compressor and the adder is shown in Figure 5.9. Note that the single delay element followed by the 2-fold compressor, designated using a dotted rectangle in Figure 5.9, is a special case of the proposed transformation shown in Figure 5.7(b) with $T = 1$, $M = 2$, and $i = 1$. Hence, this arrangement can be replaced by a unit advance element, followed by a 2-fold compressor, and a unit delay element as shown in Figure 5.9.

Similarly, the unit advance element can be moved to the inputs of the second adder. The unit advance element cancels the effect of the second unit delay element in the FIR filter. And the compressor can be moved to the input of the second adder in a similar fashion. After the conversion, we end up with the diagram shown in Figure 5.10.

The diagram in Figure 5.10 consists of two components: a polyphase decomposition unit and a computation unit. The polyphase decomposition unit, designated with a dotted rectangle in Figure 5.10, performs serial to parallel conversion, meaning that it receives two samples from the input sequence in a serial fashion and delivers them in parallel to the computation unit. The computation unit is made of two pipeline stages of computations and each stage consists of

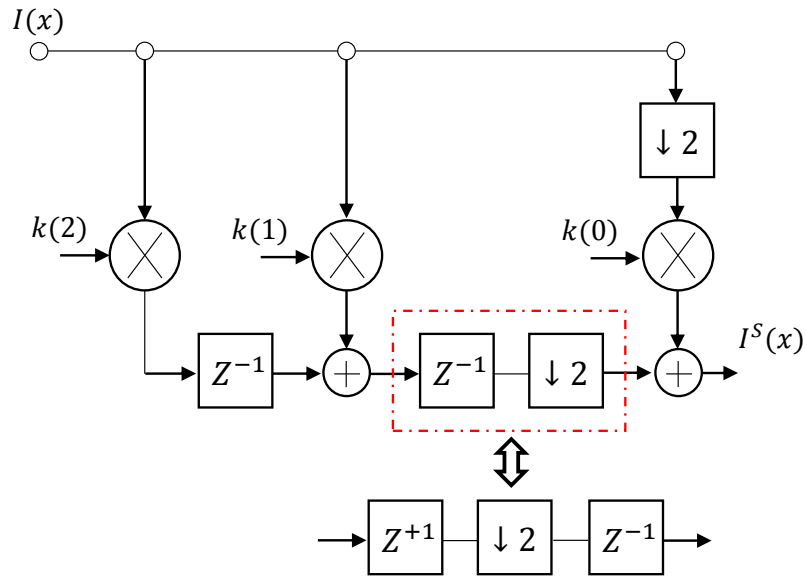


Figure 5.9: Partially transformed 1-D convolution with a stride of 2

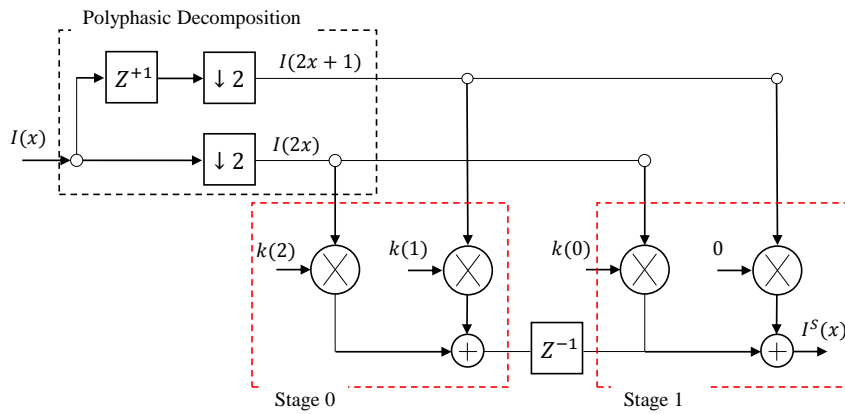


Figure 5.10: Fully transformed 1-D convolution

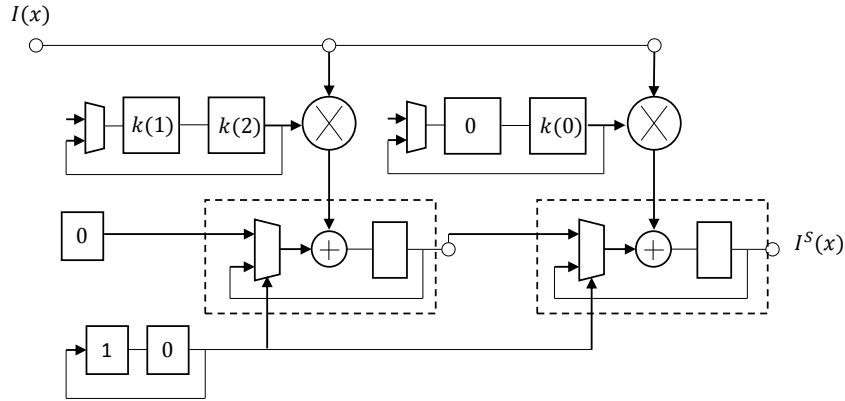


Figure 5.11: A minimal hardware implementation for a 1-D convolution filter with a size of 3 and a stride of 2

two multiplications and two additions. We designated each stage with a dotted rectangle in Figure 5.10. It is worth noting here, that the multipliers in each stage cannot operate on each clock cycle, due to the serial-to-parallel converter which delivers 2 samples every other clock cycle. Hence, in the final transformation, we remove the serial to parallel converter and collapse the multipliers that are within the same stage into a single multiplier. The resulting block diagram is shown in figure 5.11, and can be directly mapped into a hardware implementation. Note that the convolution weights shown in figure 5.11 are delivered to the multipliers in a pre-defined pattern that depends on the convolution's mask size, stride configuration, and padding parameters. We employed shift registers (shown in figure 5.11) to store the parameters on the FPGA. The shift registers exhibit a control unit that ensures its proper and synchronized operation. Moreover, a weight-loading mechanism was implemented to enable an external circuit to load a new set of weights whenever a new convolution operation is scheduled.

The hardware implementation shown in figure 5.11, consists of 2 hardware multipliers, 2 adders, 2 multiplexers, and 2 registers. Compared to the direct

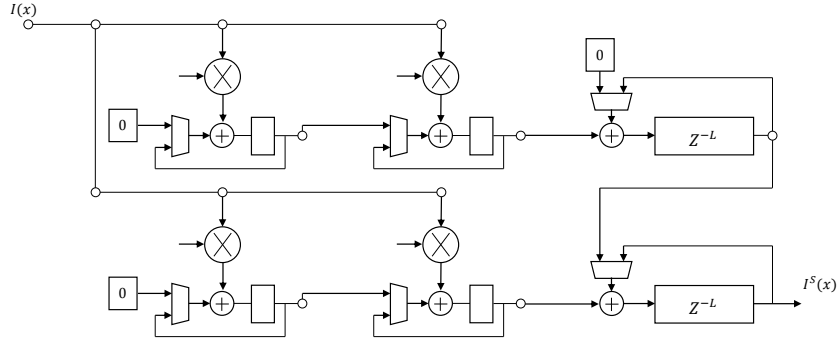


Figure 5.12: A minimal hardware implementation for a 2-D convolution filter with a size of 3×3 and a stride of 2

implementation, shown in figure 5.8, this reduced implementation was able save 1 hardware multiplier. Similarly to what we did in section 5.3.2, we can extend the 1-D convolution to 2-D. Figure 5.12, shows a 2-D convolution filter with a mask size of 3×3 and a stride of 2 in each direction.

A hardware implementation of the 2-D convolution filter shown in Figure 5.12 takes 4 multipliers, 4 adders, 4 registers, 2 delay lines, 2 multiplexers, and some control circuitry to control the multiplexers. Note that a direct implementation, i.e., an implementation like the one in figure 5.8 but for a 2-D filter will cost 9 multipliers, 6 adders, and 2 delay lines. The same methodology can be used for designing convolution filters with different sizes and stride configurations. In general, a hardware realization of a 1-D convolution filter with a filter mask of size P a horizontal stride configuration s_h , a horizontal padding configuration p_h , and designed according to the methodology proposed in this section would result in L_h adders, multipliers, registers and multiplexers, where L_h is defined as follows:

$$L_h = \left(\left(\left\lceil \frac{P - p_h}{s_h} \right\rceil + \frac{p_h}{s_h} - 1 \right) + 1 \right) \quad (5.10)$$

Similarly, a hardware realization of a 2-D convolution filter with a filter mask of size $P \times Q$, a horizontal stride configuration s_h , a vertical stride configuration s_v , a horizontal padding configuration p_h , a vertical padding configuration p_v would result in L multipliers and registers, $(L_h + 1) \times L_v$ adders and multiplexers, L_v delay lines, where L , L_h , and L_v are defined as follows:

$$L = L_h \times L_v = \left(\left\lceil \left\lceil \frac{P - p_h}{s_h} \right\rceil + \frac{p_h}{s_h} - 1 \right\rceil + 1 \right) \times \left(\left\lceil \left\lceil \frac{Q - p_v}{s_v} \right\rceil + \frac{p_v}{s_v} - 1 \right\rceil + 1 \right) \quad (5.11)$$

Coefficient Delivery Pattern

In the previous section, we employed the identity shown in figure 5.7(b) to design efficient dataflow graph representations for convolution filters with arbitrary stride configurations. This method, however, requires delivering the convolution weights to multipliers in a pre-defined pattern that depends on the convolution mask size, stride, and padding parameters. In this section, we call the forenamed pattern, *Coefficient Delivery Pattern*, and we derive a systematic methodology for obtaining this pattern for any 2D-convolution filter.

To simplify the derivation of the *Coefficient Delivery Pattern*, we define a synchronous data flow operator that we call *Coefficient Scrambler*. A 1D- *Coefficient Scrambler* is an actor object that operates on an input convolution mask to produce the desired coefficient pattern for a 1D- convolution filter with arbitrary strides; similarly, a 2D- *Coefficient Scrambler* is used to produce the desired pattern but for a 2D-convolution filter. The *Coefficient Scrambler* object has multiple attributes: (1) a mask size attribute, (2) a stride attribute, (3) a padding attribute, and (4) a repetition attribute. Figure 5.13 depicts a 1D- *Coefficient Scrambler* object acting on a convolution kernel to produce the desired coefficient

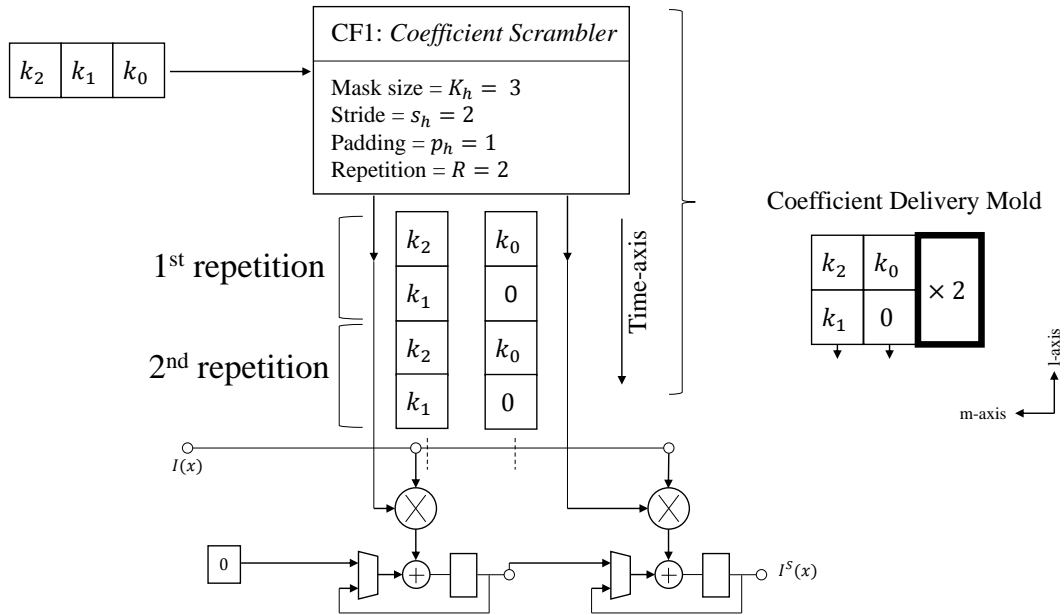


Figure 5.13: Coefficient Scrambler object feeding a 1D- filter with a mask size 3, stride 2, and padding 1

delivery pattern for a convolution filter with a mask size 3, a stride configuration 2, and a padding parameter 1. Generally, a 1D- *Coefficient Feeder* has one input port and L_h output ports, where L_h is as defined in the previous section; for each mask it receives, it produces R repetitions of its *Coefficient Delivery Pattern*.

The *Coefficient Delivery Pattern* obtained in figure 5.13 is recapitulated by, what we refer to as, the *Coefficient Delivery Mold* or CDM for short (c.f. figure 5.13). Consequently, for a certain arbitrary 1D- convolution filter, the correct *Coefficient Delivery Pattern* can be determined by properly populating a *Coefficient Delivery Mold* (CDM). As a rule of thumb, a CDM has s_h rows and L_h columns. Given a 1D- convolution with a mask $K = [k_{P-1}, k_{P-2}, \dots, k_0]$, stride s_h , padding parameter p_h , the CDM can be populated according to the following rule: if we denote the element at row l and column m of the CDM matrix as $C_{l,m}$. then

$$C_{l,m} = \begin{cases} k_r, & 0 \leq r \leq P - 1 \\ 0, & \text{otherwise} \end{cases} \quad (5.12)$$

$$r = \left(m - \left\lceil \frac{P - p_h}{s_h} \right\rceil \right) \times s_h + l + P - p_h \quad (5.13)$$

The repetition attribute of the *Coefficient Scrambler* R is also reflected in the *Coefficient Delivery Mold*; it appears at the right of the CDM matrix (c.f. figure 5.13) and can be obtained using equation 5.14. S_I , here, is the length of the input signal i.e., the length of the input feature map. Notice that we defined S_I as length instead of size, since the input feature here is a line rather than a rectangular structure.

$$R = S_I + 2 \times p_h - P + s_h \quad (5.14)$$

The concept of the 2D- *Coefficient Scrambler* is just like its 1D- counterpart, in the sense that it has a *Coefficient Delivery Mold*; we call it 2D- CDM. Figure 5.14 shows a 2D- *Coefficient Scrambler* feeding a 2D- FIR convolution filter with a mask of size 3×3 , stride 2 and padding 1. In this example, the Coefficient Scrambler has two interfaces; one interface for each row of multipliers in the convolution circuit. The 2D- *Coefficient Delivery Mold* consists of a matrix of 1D- sub-molds. In general, a 2D- CDM, has s_v rows of 1D- sub-molds, and L_v column of 1D-molds.

Given a 2D- convolution with a 2D- mask K , horizontal stride s_h , vertical stride s_v , horizontal padding parameter p_h , vertical padding parameter p_v , the 2D- CDM may be populated according to the following rule: If we denote $C_{t,v,l,m}$ the element at row l and column m in the 1-D mold, which is at row t and column

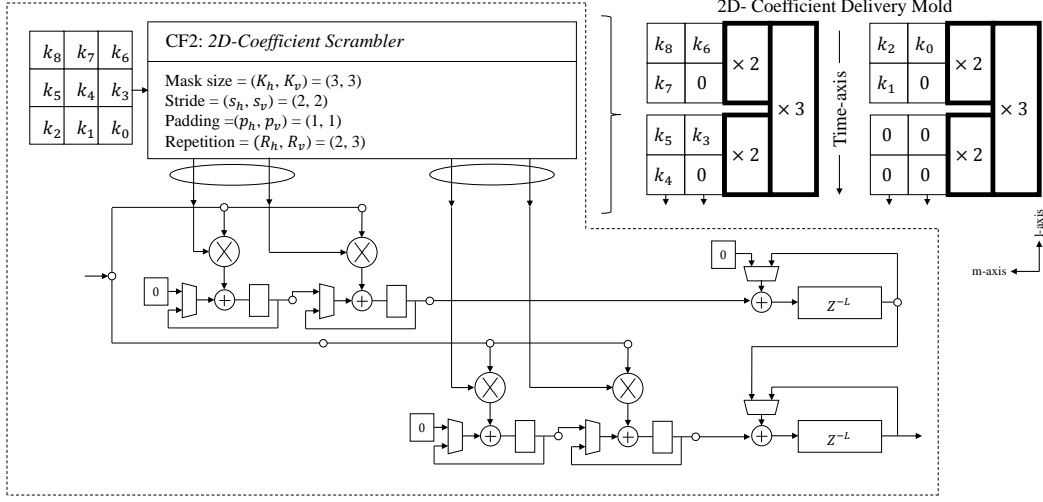


Figure 5.14: A 2D- Coefficient Scrambler object feeding a 2D- filter with a mask size 3×3 , stride 2, and padding 1

v in the 2D- CDM, then

$$C_{t,v,l,m} = \begin{cases} k_r^{r'}, & 0 \leq r \leq P-1, \text{ and } 0 \leq r' \leq Q-1 \\ 0, & \text{otherwise} \end{cases} \quad (5.15)$$

$$r = \left(m - \left\lfloor \frac{P-p_h}{s_h} \right\rfloor \right) \times s_h + l + P - p_h \quad (5.16)$$

$$r' = \left(v - \left\lfloor \frac{Q-p_v}{s_v} \right\rfloor \right) \times s_v + t + Q - p_v \quad (5.17)$$

$$K = \begin{bmatrix} k_{P-1}^{Q-1} & k_{P-2}^{Q-1} & \dots & k_0^{Q-1} \\ k_{P-1}^{Q-2} & k_{P-2}^{Q-2} & \dots & k_0^{Q-2} \\ \vdots & \vdots & \ddots & \vdots \\ k_{P-1}^0 & k_{P-2}^0 & \dots & k_0^0 \end{bmatrix}. \quad (5.18)$$

There are two repetition attributes for the 2D- *Coefficient Scrambler*: R and R' and they are reflected in the 2D-CDM as shown in figure 5.15. R and R' may

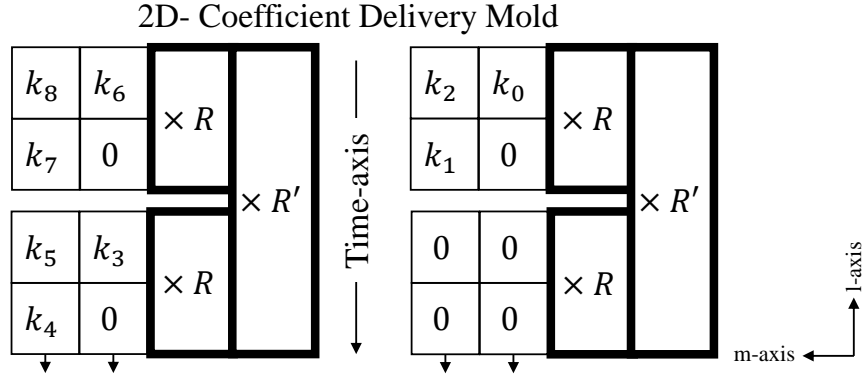


Figure 5.15: A 2D- Coefficient Scrambler object

be obtained using equations 5.19 and 5.20. S_I , here, is the width of the input image, and S'_I is its height. Note that $R \times R'$ is equivalent to the latency of convolving the input image with the convolution mask.

$$R = S_I + 2 \times p_h - P + s_h \quad (5.19)$$

$$R' = S'_I + 2 \times p_v - Q + s_v \quad (5.20)$$

Analyzing the Effect of the Reduction Scheme

A single 2D convolution filter designed according to the reduction scheme proposed earlier provides a certain level of performance or latency, while occupying a predictable amount of FPGA resources. As mentioned in section 5.3.2, the filters designed here, can process at least one input sample on every clock cycle. Consequently, a filter can process an input feature map of size $N \times N$ in around $N \times N$ clock cycles. Table 5.3 shows the amount of resources or silicon area occupied by different convolution filter designs with and without employing

Table 5.3: Hardware cost of different 2-D FIR filters with strides

FIR window size and stride	Latency (clock cycle)	Resource Count/Area-latency product				Reduction in area-latency product
		Without reduction		With reduction applied		
		Multipliers	Resources × latency	Multipliers	Resources × latency	
3x3, stride = 2	$N \times N$	9	$9(N \times N)$	4	$4(N \times N)$	$2.25 \times$
5x5, stride = 2	$N \times N$	25	$25(N \times N)$	9	$9(N \times N)$	$2.78 \times$
7x7, stride = 2	$N \times N$	49	$49(N \times N)$	16	$16(N \times N)$	$3.1 \times$
11x11, stride = 4	$N \times N$	121	$121(N \times N)$	9	$9(N \times N)$	$13.4 \times$

the reduction scheme. The table also shows the latency of computing the 2D convolution of an input feature map of size $N \times N$. Note here that we only report multiplier count as a rough estimation of the silicon area. Table 5.3, shows that up to 13 time-reduction in the area-latency product can be achieved for filters with a mask of size 11×11 and a stride of 4. In general, the reduction in area-latency becomes more pronounced with convolution filter designs with larger stride configurations.

Although the proposed reduction scheme resembles the Winograd technique for the fast computation of the convolution operation in that both schemes can be used to reduce the number of multipliers in an implementation, there are subtle differences between the two. While the Winograd technique can be exclusively used when the convolution stride configuration is one [26], the proposed reduction scheme is tailored only for convolutions with stride configurations larger than one.

Observations

We observe that a 2D Finite Impulse Response filter with arbitrary mask sizes, padding, and stride configurations can always be resolved into three constituent parts: (1) A multiplier grid, (2) a sample accumulation grid, and (3) a line accumulation grid. Those three parts are depicted in figure 5.16. As in systolic

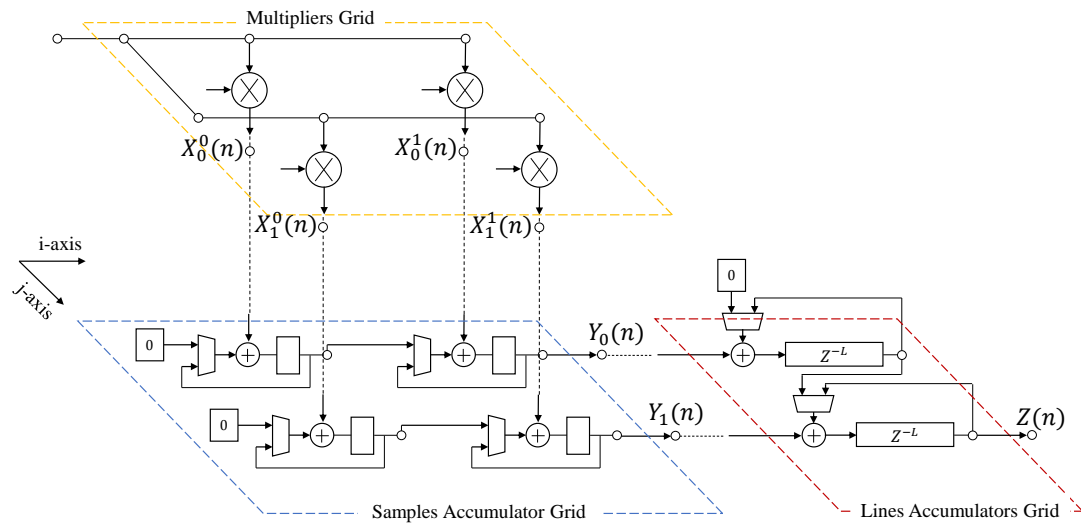


Figure 5.16: A 2D- FIR filter with mask size 3×3 and stride 2

array architectures, the sample accumulator grid is made of a regular homogeneous two-dimensional array of tightly-coupled processing elements (PEs). We will henceforth refer to these processing elements as sample-add-accumulate PEs. Similarly, the line accumulator grid is also composed of a one-dimensional vertical array of PEs that we refer to as line-add-accumulate units. Figure 5.17 depicts both the sample-add-accumulate and the line-add-accumulate PEs.

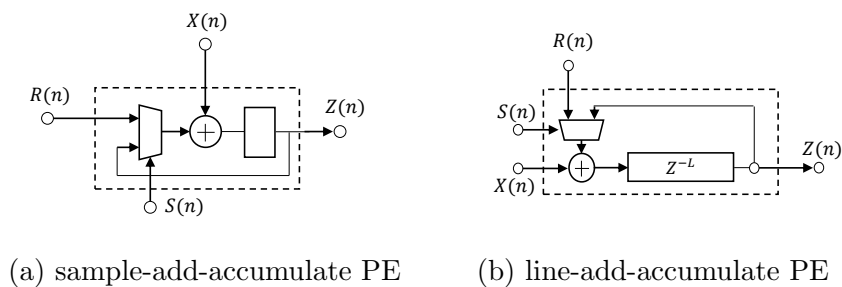


Figure 5.17: The sample and line accumulation grids that are common among all 2D-fir filters

Both the sample accumulation, and the line accumulation grids can be modeled as Multiple-Input-Multiple-Output systems. We denote those systems as

T_1 and T_2 . The output of the sample accumulation grid i.e., system T_1 can be formulated mathematically as follows:

$$Y_j(n) = \sum_{i=0}^{s_h-t} X_j^{L_h-1}(n-i) + \sum_{r=0}^{L_h-2} \sum_{i=0}^{s_h-1} X_j^r(n-s_h \times (L_h-r-1) + i) \quad (5.21)$$

Here t is the smallest positive integer such that $n+t = K \times s_h$, where K is a positive integer. L_h is the number of multipliers along the i-axis, and s_h is the horizontal stride configuration. Similarly, the output of the Line accumulator grid can be formulated mathematically as:

$$Z(n) = \sum_{i=0}^{s_v-l} Y_{L_v-1}(n-i \times S_0) + \sum_{r=0}^{L_v-2} \sum_{i=0}^{s_v-1} Y_r(n-s_v \times (L_v-r-1) \times S_0 + i \times S_0) \quad (5.22)$$

Here l is the smallest positive integer such that $\lfloor n/s_0 \rfloor + l = K \times s_v$, where K is a positive integer. L_v is the number of multipliers along the j-axis, s_v is the horizontal stride configuration, and S_0 is the size of the line that results from each individual 1D-Convolution .i.e, $S_0 = \frac{S_I+2*p_h-P}{s_h} + 1$. We can prove that both the sample and the line accumulation grids are linear time invariant systems. The linearity property can be leveraged to aid in reducing resource utilization and building more compact efficient hardware implementations. A case where this property proves to be very useful in deriving efficient hardware architectures for convolutions, is when it is required to perform 2D-convolutions on two different input feature maps with two different convolution kernels and add the resulting output feature maps into a single feature. This case is depicted in figure 5.18, where two input images I_0 and I_1 are convolved with two different kernels K and

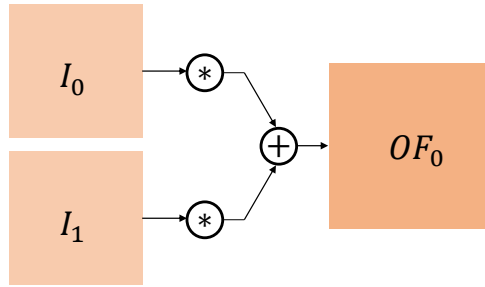


Figure 5.18: Case where the outputs of two 2D-convolution filters are combined into a single output feature

K' , and the output of the convolutions are combined into a single output feature map.

Since the sample and line accumulation grids are both linear systems, the adder at the output of both convolutions may be absorbed into the 2D-convolution operations. The convolution operations may share the same sample accumulation and line accumulation grid, as shown in figure 5.19. This sharing of resource is commonly referred to in the literature as sub-structure sharing, and it has significant impact on the resource utilization of FPGA implementations.

5.3.4 Optimizing for Finite Word-length Representation and Computation

In embedded systems where power is of major concern, the use of fixed-point arithmetic is an appealing alternative to floating-point arithmetic due to its smaller logic resource requirements. This is particularly advantageous, and at times necessary, in resource-limited FPGAs. Moreover, some applications do not require the high numerical precision of floating-point arithmetic. For instance, Deep Convolutional Neural Networks are well known for their resilience and their ability to cope with limited precision arithmetic, especially, during inference [97].

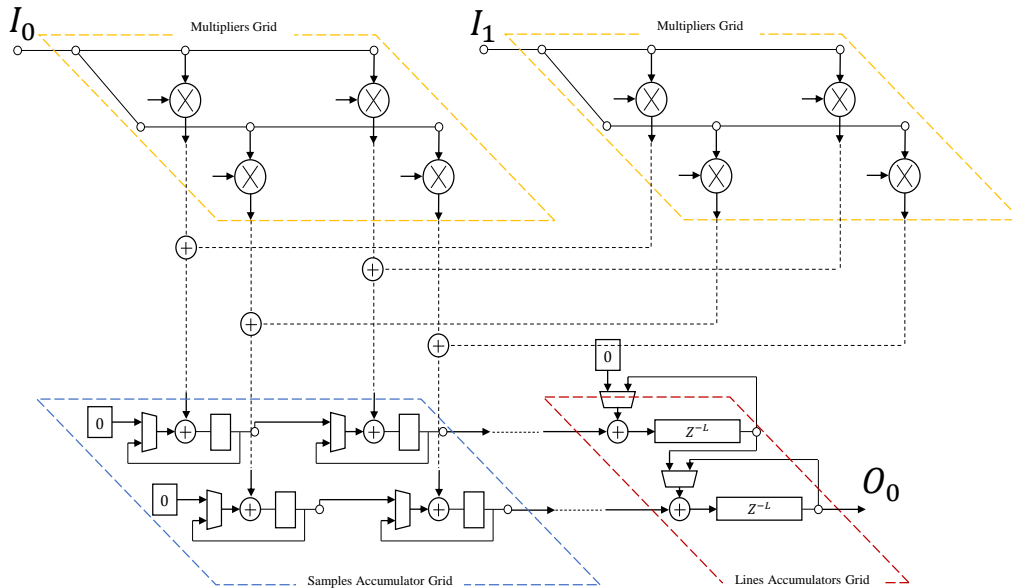


Figure 5.19: Two 2D-FIR filters sharing the same sample and line accumulate networks

As a result, to implement a resource and energy efficient hardware design, we use a limited-precision fixed-point representation. In this section, we describe the methodology, and evaluate a low precision fixed-point implementation for a deep convolutional neural network.

Effect of finite word-length representation of model parameters

Before we investigate the effect of fixed-point arithmetic, we begin by analyzing the effect of finite precision fixed-point representation of network parameters on the accuracy of the classifier. We use AlexNet to illustrate the methodology, but the same approach can be employed with any other neural network classifier. To investigate this effect, we use software-based fixed-point simulations, in which we run a fixed-point software implementation of the algorithm on a pre-determined dataset. We used a pre-trained Caffe-compatible AlexNet model obtained from

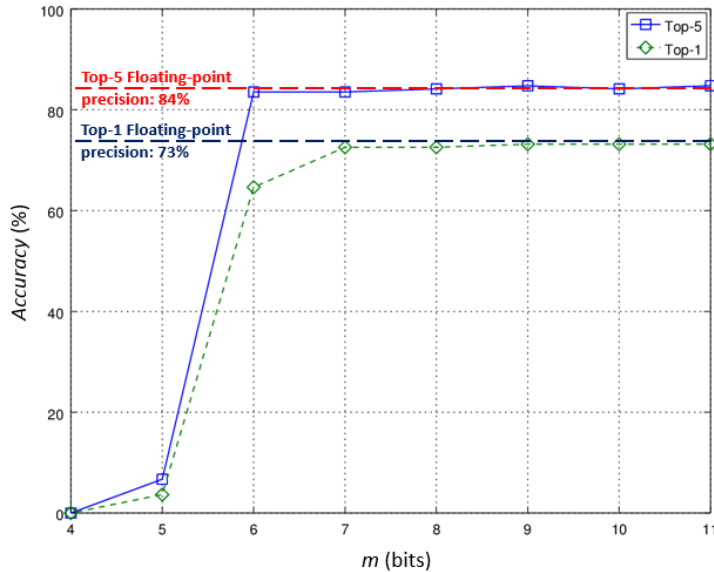


Figure 5.20: The effect of fixed-point representation of network parameters on the accuracy of AlexNet

[132]. The images used in the simulation are randomly selected from the ILSVRC-2012 dataset. We Evaluate two metrics: the top-1 and top-5 accuracies [87]. Note that, in this work we use 2’s complement format and we adopt the notation used in [24] to characterize the structure of a fixed-point representation. i.e., a (n, m) fixed-point format would denote a representation that uses n bits to represent integers, and m bits to represent fractions. We, also, observe that AlexNet parameters are always between -1 and 1, hence a suitable fixed-point representation for them is $(1, m)$.

Figure 5.20 shows how the top-1 and top-5 accuracies change with m . The top-5 and top-1 accuracies of a floating-point implementation of AlexNet are indicated by dashed horizontal lines on figure 5.20. Note that the top-5 accuracy of AlexNet with fixed-point parameters declines only when the number of fractional bits is less than 6. Given that all parameters are between -1 and 1, we can use a global fixed-point representation $(1, 7)$ for all the parameters of AlexNet without

Table 5.4: Possible fixed-point representation for every layer in AlexNet

Layer	Minimum	Peak	Representation (n, m)		Total
			n	m	$n+m$
Data	-123	151	9	-1	8
Conv1	-2524	2587	13	{3, 11, 19}	{16, 24, 32}
Norm1	0	139	9	{7, 15, 23}	{16, 24, 32}
Conv2	-923	645	11	{5, 13, 21}	{16, 24, 32}
Norm2	0	139	9	{7, 15, 23}	{16, 24, 32}
Conv3	-489	431	10	{6, 14, 22}	{16, 24, 32}
Conv4	-252	284	10	{6, 14, 22}	{16, 24, 32}
Conv5	-189	207	9	{7, 15, 23}	{16, 24, 32}
FC6	-104	69	8	{0, 8, 16, 24}	{8, 16, 24, 32}
FC7	31	18	7	{1, 9, 17, 25}	{8, 16, 24, 32}
FC8	-10	35	7	{1, 9, 17, 25}	{8, 16, 24, 32}

impacting the top-5 and top-1 accuracies.

Effect of fixed-point arithmetic

One of our aims in this work is to reduce hardware complexity by minimizing $n + m$ without impacting the accuracy of the classifier. This trade-off between accuracy and hardware complexity is best tackled using the multiple word-length paradigm [24]. In this paradigm, every computational stage in AlexNet inference is assigned a different fixed-point representation instead of a system-wide representation. The reasoning behind this paradigm is two-fold: (1) signals in different computational stages have different dynamic ranges and thus, requires different representations, and (2) every computational stage or layer in AlexNet inference contributes differently to the output stage of the classifier. Table 5.4, shows the dynamic ranges of different layers in AlexNet and n .

Note that data movement circuits, i.e., circuits that moves data between external memory and the FPGA, are much simpler when the word-length representations of data samples are multiples of 8-bits, since typical on-board memories are byte addressable. Hence, the number of fractional bits, m , is chosen in such

a way as to keep the total word-length in each layer a multiple of 8-bits.

To find the best combination of m values, we resort to fixed-point simulations. A brute force approach might consist of simulating all possible combinations of value for m and select the combination that gives the best classification accuracy. However, this approach is costly, as there are 139,968 possible combinations. We use a heuristic method to find a minimal solution in terms of hardware cost but keeps the accuracy at an acceptable range. The reasoning behind this method is that the first few layers are the most critical and hence we may start by gradually reducing the fractional part of the first layer until the accuracy starts dropping. Then we shift to the next layer and start reducing its fractional part until the accuracy starts dropping again. This procedure is repeated until we reach the last layer. Following this procedure, we found that the following combination of values of m $(-1, 3, 7, 5, 7, 6, 6, 7, 8, 9, 9)$ does not degrade the accuracy of the classifier. We have used this combination of fixed-point representations for the different layers in our minimal implementation in the Evaluation and results section.

5.4 Minimalists Accelerated Convnet System Architecture

In this section, we describe a method for mapping the filters we designed, in section 5.3, into an FPGA hardware accelerator. The proposed hardware accelerator conforms with the guidelines we proposed in chapter 3. Recall that in chapter 3, we defined the accelerator as having three kinds of stream interfaces: (1) parameter interfaces, (2) input data interfaces, and (3) output data interfaces. In this implementation, we gave the accelerator more than one parameter interface. the parameter interfaces are used to load the ConvNet model parameters from

external memory. The input data interface is used to load the input image into the accelerator, while the output data interface is used to capture the result of the classification into external memory.

5.4.1 Proposed Architecture

Figure 5.21 illustrates the general functional architecture of our proposed template accelerator. The template design of the minimalist ConvNet accelerator consists of three main types of components: (1) data-flow computational engines, (2) data movement circuitry, and (3) on-chip cache memory. The Data-flow computational engines are responsible for carrying out all the computational sub-tasks in a certain implementation of a CNN, whereas the data movement circuitry is responsible for reading data from external memory and feed it to the accelerator. The on-chip cache memory component is used to cache or buffer the intermediate data generated by the different sub-computations. We define a computational stage or sub-task as a set of transformations that starts with a convolutional or fully-connected layer, followed by an optional normalization layer and then by an optional pooling layer. This allows ConvNet inference to be modeled as a chain of computational stages. Consequently, ConvNets have two types of computational stages: convolutional and fully-connected, and, therefore, in this architecture, there are only two types of Data-flow computational engines: (1) convolutional and (2) fully-connected engines.

A typical convolutional engine is a chain of computational filters that begins with a 2-dimensional convolution filter, and is followed by an activation filter, which computes an element-wise non-linear activation function on an input feature map; a Local Response Normalization (LRN) filter, and a pooling filter. In section 5.3, we devised a systematic methodology for designing minimal hard-

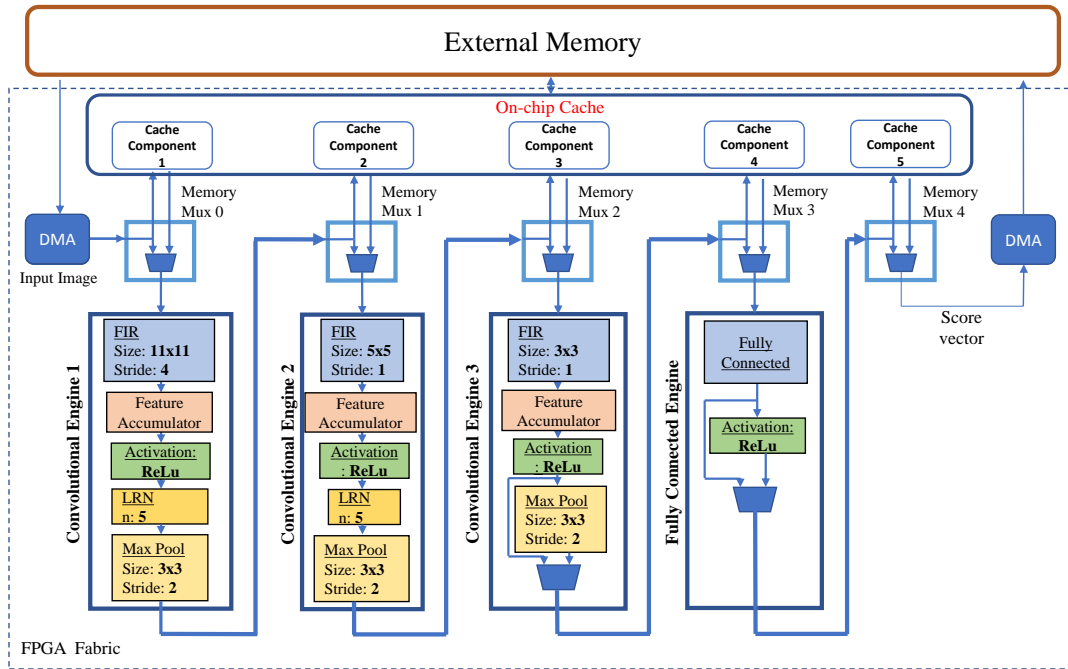


Figure 5.21: System Architecture

were realizations for each type of those filters. A Fully-connected engine, on the other hand, consists of a simple dot-product filter. This architecture enables the designer to further boost the performance of a convolutional engine by deploying multiple 2-dimensional convolution filters in one convolutional engine, and consequently multiple normalization and pooling filters. However, since we are aiming for a minimalist implementation, we limit the number of filters to one per engine.

The on-chip cache memory, shown in figure 5.21, is implemented using block-RAMs, and is organized into multiple, independently-accessible, simple dual-port RAM memories, called cache components. Each cache component acts as a ping-pong buffer. Therefore, the data-flow computational engines can access different cache components in parallel, allowing the computational engines to operate concurrently while the cache components act as buffers. This choice of cache organization allows a form of temporal parallelism or pipelining to occur,

i.e., input features can be received by each computational stage just as they are released by the previous one.

The two data movement circuits are referred to as DMAs in the figure. One DMA unit feeds input images to the pipeline while another DMA writes the resulting score vectors back to memory. Other DMA units (not shown in the figure) are also used to load convolution weights from memory to the convolution filters. Special components called Memory Muxes (c.f. Figure 5.21) are used to control the flow of data between the computational engines, the on-chip cache, and the data movement circuitry. A memory mux can operate in two different modes (c.f. Figure 5.22). In the first mode of operation, referred to as Mode A and depicted in figure 5.22(a), the mux alternates between two states: in the first state the data stream flowing into port 1 is forwarded to both ports 2 and 4 (Flow 1). In the second state, the data stream flowing into port 3, is forwarded to port 4 (Flow 2). This mode of operation is useful when the on-chip cache sub-component is not included in the design and the data-flow computational engines are supposed to buffer their output data on external memory. In figure 5.22(a), the data stream received from a previous computational stage through port 1, is simultaneously dispatched to the current computational stage, and to external memory. The memory mux then switches to the next state, where flow 2 is now active, allowing the current computational stage to receive the data that was generated by the previous stage, from external memory. In the second mode of operation, referred to as Mode B and depicted in figure 5.22(b), the data stream flowing into port 1, is dispatched to port 2, while the incoming data at port 3 is forwarded to port 4. Note that, in this mode, flows 1 and 2 are both active at the same time. This mode of operation, is useful when the on-chip cache is included in the design, since it allows the previous stage or computational engine to write

to a cache component, while the current computational engine is reading from it.

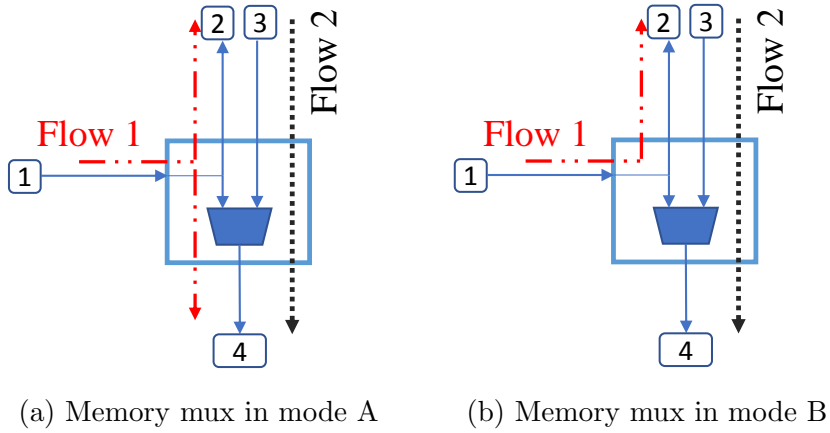


Figure 5.22: The memory mux component

Communications between different computational stages are scheduled by special control units. In fact, every memory mux has its own independent control unit, which controls its behavior and mode of operations. Equally, each cache component has its own control unit, which mediates read and write operations. The data movement circuits or DMAs, on the other hand are controlled by an on-chip processor core. And since, the DMAs are configured to operate in scatter-gather mode, the on-chip processor can chain together multiple simple DMA requests to offload multiple interrupts and consequently hide the latency of the interrupts. By controlling communications between different stages, the control units along with the on-chip processor core can handle the scheduling of computations on the computational engines. Upon scheduling a convolution operation, a DMA unit loads the weights that correspond to the scheduled convolution into the corresponding convolutional filter from external memory. Subsequently, the corresponding memory mux is scheduled to read the corresponding feature maps from a cache component and feed them to the convolutional engine.

5.4.2 Modularity and Portability

The rapid evolution of Convolutional Neural Network architectures, and the wide variety of FPGA devices in the market, promote designs with two important properties: (1) Modularity, and (2) Ease of portability between different FPGA vendor platforms. To achieve a modular design that can support different types of ConvNet layers while ensuring efficient resource utilization, we propose the modules to be relatively small. In our design, the modules are composed of computational filters that can be interconnected to build an entire accelerator design. In section 5.3, we provide a methodology for designing computational filters. These model elements can be easily configured through HDL parameters (e.g. layer type, mask size, stride, etc..) to meet the needs of specific ConvNet implementations and thus they can be reused to implement a wide variety of ConvNet architectures. Although we used these model elements to implement AlexNet, the modular nature of the design allows the easy implementation of other ConvNets as well. In the context of targeting resource-restricted FPGAs, the modular nature of our design strategy allows us to easily remove model elements from an overall design without affecting the flow of data within it, and thus when targeting the implementation of a specific ConvNet that does not contain a particular layer type, the model element that correspond to this particular layer can be removed and thus further saving FPGA resources. It is worth noting that our approach is different from previous modular ConvNet designs, where the modules are composed of relatively large blocks of logic resources and aim at achieving specific functionalities instead of smaller blocks of filters.

To ease the portability between two different FPGA vendor platforms, we wanted to make sure that our HDL implementation is not specific to one FPGA product, and that the design is portable across two competing FPGA vendor

platforms, namely Intel/Altera and Xilinx. Previous approaches [105,106] achieve similar portability by implementing their designs in behavioral VHDL/Verilog, and rely on the design/synthesis tools to resolve the micro-architectural differences and infer the device-specific components for each target device (BRAMs, DSPs, ...). The problem with this approaches is that synthesis tools do not have information about the purpose or intent of the design, and thus may try to optimize for area and performance by inferring device primitives. As a result, the tools may misplace the individual hardware blocks (e.g. BRAMs and LUTs) in certain sub-components of the design. To address this limitation of proper placement, we relied on instantiating the device-specific hardware blocks needed in each computational filter rather than simply relying on the tools to infer these blocks while keeping track of the similarities and differences between the two platforms. For example, due to the dimensions of the ConvNet feature maps involved in the computations in AlexNet, normalization filters are best implemented using BRAMs, whereas max pooling filters are best implemented using LUTRAMs in Xilinx FPGAs and MLABs in Cyclone V. To achieve the desired portability, we developed a hierarchical design that decouples the device-specific features needed to meet the low resource utilization target from the rest of the design, and we used VHDL generics to seamlessly instantiate and integrate the device-specific components for the target device. Using this decoupling in our hierarchical design, we were able to successfully implement our ConvNet accelerator design on two architecturally different FPGA devices, the Xilinx Zynq 7020 [33] and Intel Cyclone V [34], using two different logic synthesis tools. Our design can be easily ported to other devices by editing a small number of VHDL source files.

5.4.3 Design Entry and Implementation

High level synthesis tools such as Xilinx HLS [114] and Intel’s OpenCL framework [115] raise the design abstraction level and allow for an easy and rapid exploration of the application design space [118, 133]. Although High-level Synthesis tools provide an attractive solution to designing FPGA-based hardware accelerators, their limitations are well known and understood in the literature. According to Bailey [118], high level synthesis tools have many limitations: (1) Random C-like codes does not map very well to hardware implementations, and thus algorithms must be re-written in a particular style to enable the synthesis tools to exploit parallelism, (2) Algorithms that use pointers must be restructured to use array references, since the use of pointers obscures the data-flow properties of the algorithm, (3) recursion, which is extensively used in procedural-based programming, does not map well to hardware implementations, (4) high-level synthesis tools does well when scheduling and pipelining sequences of operation, but struggles with complex synchronizations, (5) the RTL code produced by high-level synthesis tools is not human readable and thus it is hard to perform RTL verification, or debug and modify the resulting RTL code, (6) the best representations for hardware accelerators are not necessarily based on high-level languages, and some computations are best described using data-flow-based block diagrams (e.g. Image filtering, 1D- and 2D- Fast Fourier Transforms, ...).

As mentioned in section 5.2, Xilinx HLS has previously been employed to implement ConvNet accelerators on FPGA platforms in [25, 116, 117], while Intel’s OpenCL was employed in [26]. The use of high level synthesis tools in [25, 26, 116, 117] assisted the designers in exploring the ConvNet inference design space, and alleviated the effort of implementing the resulting accelerators, as these tools can automatically analyze the structure of the high-level language

specifications of the different sub-computations involved in a ConvNet and extract the necessary Data and Control paths. Implementing ConvNet inference in VHDL or Verilog was the subject of [105] and [106]. In their work, however, they did not address the challenge of implementing Complex ConvNet inference workloads on resource-constrained FPGA platforms. In contrast with the works in [25, 26, 105, 106, 116, 117], we approach the problem of mapping complex ConvNet inference workloads to low-end and resource-constrained FPGA platforms that are suited for IoT deployment, thus understanding the tradeoff between area (or resources occupied by the accelerator) and latency is important. As mentioned in section 5.3.2, our approach relies on modeling the different sub-computations involved in a ConvNet inference as DSP-based block diagrams or signal processing chains. We also derived a graph-based method for mapping the aforementioned DSP-based block diagrams to minimal-resource filter realizations that can be directly implemented in hardware. These filter realizations are best described using structural VHDL/Verilog, rather than procedural languages such as those involved in high-level synthesis tools; moreover, with VHDL/Verilog we can have more control over the resources allocated in the design (e.g. number of multipliers, adders, and BRAMs). Consequently, we implemented our design using a combination of both VHDL and Verilog. We used Verilog to instantiate the device-specific components (e.g. BRAMs, LUT RAMs, and DSP units), and VHDL to assemble the device-specific components together and to implement the control path. As mentioned earlier, the VHDL design exhibits a hierarchical structure that decouples device-specific features from the rest of the design, and this design choice eases the portability of the design across different architecturally different FPGA devices. The benefits of employing VHDL/Verilog to implement the filter structures derived in section 5.3.2 can be summarized as

follows:

1. Implementing the design in structural VHDL/Verilog allows for more control over the exact placement of resources (e.g. DSP units, BRAMs, and LUTRAMs). As mentioned earlier, the resulting filter structures derived in section 5.3.2 can be mapped directly into hardware implementations, and they are best described and implemented using structural VHDL/Verilog.
2. The DSP-based representations of inference sub-computations allow us to further improve performance or reduce area (FPGA resource) requirements by employing other DSP-based optimizations such as the strength reduction schemes described in reference [104]. We left employing these optimization schemes to a future work.
3. In our approach, we are able to derive a relationship between the amount of FPGA resources occupied by the accelerator and the computational latency. The amount of resources occupied by a convolution filter with arbitrary hyper-parameters (i.e., mask size, stride, and padding) was derived in section 5.3.3 equation (5.11), while the latency of computing a single output feature map was derived in section 5.3.3 equations (5.19) and (5.20). This relationship is vital for understanding the tradeoff between area (resources occupied by the accelerator) and performance. In chapter 6 section 6.2.2, we employed this relationship to estimate the performance of an FPGA platform when implementing a particular ConvNet inference workload and when all the available resources are employed.

Table 5.5: Evaluation Platforms and their characteristics

Platform		ZedBoard™	Cyclone® V Dev-Kit
Manufacturer		Avnet™	Intel®/Altera®
Board	On-board RAM	512 MB	384 MB + 512 MB
Resources	FPGA	Zynq 7020	Cyclone V GT D9
FPGA	Block RAMs	140 (5,160 Kb)	1220 (12,200 Kb)
Resources	DSP Units	240 DSP48E	342 DSP
	Lookup tables	53,200	113,560
Process Technology		28nm HPL	28nm LP

5.5 Evaluation and Results

5.5.1 Evaluation and Experimental Setup

Evaluation Platforms

As mentioned earlier, in this work, we aim at deploying ConvNets on low-end resource-limited FPGA platforms. For this purpose, we decided to deploy our accelerator design described in section 5.4, on two low-end FPGA platforms: The Xilinx Zynq 7020 SoC device [33], and the Intel Altera Cyclone V GT device [34]. We used the “Zynq Evaluation and Development” Board (ZedBoard) from Avnet [35] to target the Zynq 7020 FPGA. For the Cyclone V FPGA, we used the Cyclone V Development Kit [36]. Table 5.5, summarizes the available resources and the key differences between the two platforms.

Evaluation Network and Dataset

We deployed a pre-trained Caffe-compatible AlexNet model from the Caffe model Zoo [132]. Note that Caffe [92] is a deep learning software framework developed by the UC Berkeley Artificial Intelligence Lab. This framework is typically used to train and deploy deep neural networks. The pre-trained AlexNet model is trained on the ILSVRC-2012 [87] dataset. We sample a random selection of

images from the ILSVRC-2012 validation dataset to test our accelerator.

Evaluation Metrics

To evaluate the performance of our design, we measured seven different performance metrics: FPGA resource utilization, end-to-end latency measured in terms of milliseconds (ms), throughput measured in terms of frame per second (frame/sec), performance measured in terms of Giga Operations per Second (GOPs), Energy efficiency measured in terms of Joules per frame (Joule/frame), Energy efficiency per cost measured in terms of Frame per Joule per Dollar ($[\text{frame/Joule}] / \text{Dollar}$), and finally the total board power consumption measured in terms of Watts.

5.5.2 Results

Performance of Basic Design

First, we implemented the architecture shown in Figure 5.21, but without including the on-chip cache memory which buffers the intermediate feature maps on FPGA Block-RAMs. The buffering scheme was presented in section 5.3.4. The implementation employs the fixed-point representation scheme derived in section 5.3.4. We collected resource utilization, performance and energy results on the two low-end FPGA platforms described in 5.5.1. Table 5.6 shows the resource utilization results on both platforms. Note that the resource utilization percentages are on average around 30% of all the available resources on the FPGA fabric in both platforms.

The performance and energy per frame results are shown in Table 5.7. Note that the execution time of AlexNet is around 1.3 seconds on the ZedBoard and

Table 5.6: Resource Utilization for AlexNet on ZedBoard and Cyclone V

Platform	ZedBoard™		Cyclone® V	
	Count	Percentage	Count	Percentage
DSP	55/220	25%	53/342	15%
BRAM	49/140	35%	322/1220	26%
Look-up Tables	16,536/53,200	31%	20,898/113,560	18%
Flip Flops	31,976/106,400	30%	83592/454,240	18%

Table 5.7: Performance and Energy Results of AlexNet implementation without the on-chip cache.

Platform	ZedBoard™	Cyclone® V Dev-
Operating Frequency	100 MHz	150 MHz
Latency (mSec)	1,332.4 ms	1,666 ms
Throughput (frame/sec)	0.75 frame/sec	0.6 frames/sec
Energy (Joule/frame)	1.918 Joule/frame	1.899 Joule/frame
Performance (GOPs)	0.543 GOPs	0.434 GOPs
Performance Density	3.28×10^{-5} GOPs/LUT	2.08×10^{-5} GOPs/LUT

around 1.6 seconds on the Cyclone V dev-kit. This is mainly, because every convolutional layer needs to reuse the input feature maps multiple times while computing the output feature maps. Moreover, without the on-chip cache component, input features must be buffered on the External Memory (refer to Figure 5.21) which is much slower than the on-chip cache component.

Performance of the Accelerator when Caching is Enabled

Table 5.8 shows the measured latency of each computational stage in AlexNet, when the on-chip cache memory, described in section 5.4, is included in the design. Note that the end-to-end latency is reduced from 1.3 sec to 0.6 sec on the ZedBoard, and from 1.6 sec to 0.4 sec on the Cyclone V development kit. The reason for this reduction in the individual latencies of each stage is that enabling the on-chip cache memory allows the intermediate input feature maps

Table 5.8: Latency of individual compute stages of AlexNet with the on-chip cache included.

Platform	ZedBoard™	Cyclone® V Dev-
First Stage	148 ms	99 ms
Second Stage	89 ms	60 ms
Third Stage	166 ms	110 ms
Fourth Stage	124 ms	83 ms
Fifth Stage	83 ms	55 ms
FC6	8.8 ms	7.5 ms
FC7	3.95 ms	3.35 ms
FC8	0.96 ms	0.81 ms
Total	623.71 ms	418.66 ms

to be buffered in the on-chip cache sub-components (refer to section 5.4), and thus every convolutional engine can now consistently read one sample from the cache sub-components on every clock cycle (refer to figure 5.21 and to section 5.4); Consequently, the latency of convolutional stages is equal to $N \times N \times I_f \times I_o / G$ clock cycles, where $N \times N$ is the size of the input features, I_f the number of input features, I_o the number of output feature maps, and G the group hyper-parameter; and the latency of fully-connect stages is equal to $N_I \times N_O$ memory cycles, where N_I is the number of input neurons, and N_O is the number of output neurons. Moreover, as mentioned section 5.4, the on-chip cache sub-components are configured as independently-accessible simple dual-port RAM memories. this choice of cache organization allows the different data-flow computational engines to operate in parallel. This temporal parallelism can be exploited to establish an execution pipeline made of the different computational stages of AlexNet. Consequently, the accelerator can run at the rate of the slowest computational stage i.e., at 6 frames/sec on the ZedBoard and at 9 frames/sec on the Cyclone V.

Table 5.9: Comparison between our implementation and other work in the literature.

Reference	Zhang et al [25]	Aydonat et al [26]	Our design with on-chip cache included	
			ZedBoard™	Cyclone® V
FPGA Chip	Virtex7 VX485T	Arria 10 -1150	Zynq ZC7020	Cyclone V GT D9
Frequency	100 MHz	303 MHz	100 MHz	150 MHz
Max DSP capacity	2800 DSP	1518 DSP	240 DSP	342 DSP
Performance	61.62 GOPS	1382 GOPS	4.358 GOPS	6.516 GOPS
Frame/sec	46 (Conv Layers)	1020	6	9
Latency (msec)	21.61	0.98	621.23	418.66
Top-5 Accuracy	Not Reported	79%	83.58%	83.58%
Energy/frame	0.4 J/image	0.043 J/image	0.239 J/Image	0.126 J/image
Frame/Joule	2.5 image/J	23.2 image/J	4.18 image/J	7.9 image/J
Frame/Joule/\$	7.15×10^{-4}	5.17×10^{-3}	9.31×10^{-3}	4.42×10^{-3}
Board Power	18.61 Watts	45 Watts	2.95 Watts	8.5 Watts

Comparison to other implementations

In this section, we make a quantitative comparison between our design and two other FPGA-based accelerator implementations from the literature that are based on AlexNet. We reported performance, accuracy, board power, energy and energy efficiency per cost measurements in table 5.9. The first design we are comparing against is from the work of Zhang et al [25], which deploys their accelerator on a high-end Xilinx Virtex7 FPGA. This implementation employs a MicroBlaze soft processor core to assist with accelerator startup, communication with the host computer, and with time measurements. The Second implementation is from the work of Aydonat et al [26], which is the current state-of-the-art. The platform used in [26] is a high-end Arria 10 -1150 platform that has a PCIe interface connecting it to a host computer, which controls all memory transfers and kernel executions using the Intel SDK for OpenCL.

Discussion and Analysis of Results

The minimalist accelerated ConvNet system architecture shown in figure 2, was implemented on two different resource-constrained FPGA platforms: The ZedBoard and the Cyclone V Development kit. In terms of suitability for deployment in IoT device, the designs in [25] and [26] use a relatively large number of computational resources and cannot fit on low-end resource-constrained FPGA devices needed for IoT deployment, but rather they fit for data-center deployment. Furthermore, deploying the FPGA devices presented in references [25] and [26] in IoT devices is not an option since they consume a lot of power (18.61 Watts in [25] and 45 Watts in [26]) which can quickly drain the small battery that powers the IoT device. On the other hand, both of our implementations, i.e., our implementation on the ZedBoard and on the Cyclone V platform, consumes significantly less power (2.95 Watts and 8.5 Watts, respectively) making them better suited for IoT applications with limited power sources than [25] and [26]. Furthermore, in order to reflect the tradeoff between performance and platforms' costs, we compare our implementations to references [25] and [26] using the "Frame per unit Energy per Cost (measured in frames/joule/dollar)" as our performance metric. Our ZedBoard implementation achieves significantly better than both [25] and [26] in terms of energy efficiency per cost which is measured in terms of Frames per Joule per Dollar, and which reflects the good tradeoff between performance and the costs of ZedBoard. Our Cyclone V implementation also achieves better performance than [25], and comparable performance to [26]. Finally, it is worth mentioning that, as indicated in table 5.6, our implementations on both resource-constrained platforms uses only 25% of the DSP resource on the ZedBoard and 15% of the DSP resources on the Cyclone V. This provides an opportunity to further improve performance by leveraging additional resources to improve frame

rates; or reduce power and energy consumption by migrating to smaller devices. However, we leave the exploration of these two options for a future work.

5.6 Conclusion

This paper presents an efficient methodology for mapping large convolutional neural networks to resource-constrained FPGA platforms targeted at IoT deployment; we demonstrated this framework by building a minimalist accelerator design for AlexNet. Our results demonstrated the success of addressing the design challenges and achieving low (30%) resource utilization for the lower end FPGA platforms: Zedboard [35] and Cyclone V [36]. The design overcame the limitation of designs targeted for high end platforms which cannot fit on low end IoT platforms [101–103]. Furthermore, our design showed superior performance results (measured in terms of performance/watt/dollar) compared to high end optimized designs (9.3110^{-3} Frame/J/\$ compared to 5.1710^{-3} for the state-of-the-art [26]). The designs also showed accurate results consistent with non-accelerated designs, where images from the ILSVRC-2012 dataset were classified with a top-5 accuracy of 83.58% (compared to 79% in [26]) while achieving the low-end FPGA benefits of improved energy efficiency per cost at 9.3110^{-3} Frame/J/\$ and a frame rate of up to 9 frames/sec.

Chapter 6

Conclusions and Future Perspectives

6.1 Conclusions

In chapter 3, we developed a Network Attached Accelerator (NAA) system architecture that allows deploying FPGA platforms in a server cluster data center environment. An NAA compute node, which hosts an FPGA device, may be directly attached to the facility's network infrastructure without the need for a CPU-bound x86-64 server node to host it. This choice of deployment model improves scalability, since a data center operator may easily add more FPGA devices by simply attaching more NAA compute nodes to the network infrastructure. Our proposed NAA compute model allows the cluster's master node to dynamically build a complex logical network of FPGA-based hardware accelerators that spread across multiple NAA compute nodes. On the level of a single NAA compute node, the reconfigurable component may host multiple FPGA-based accelerators simultaneously, where every accelerator has three types of streaming interfaces: (1) input, (2) output, and (3) parameter interfaces. Moreover, every FPGA-based accelerator is allowed to access memory during its operation through an optional dedicated memory interface.

The processor component of the NAA compute node runs a monitor program, that we call firmware. The firmware implements supervisory, communication, and internal routing functions. Finally, the firmware allows an extended form of the Spark cluster computing platform to target NAA compute nodes and to use them to establish complex compute pipelines. Although we didn't cover the necessary changes that should be made to the Spark middleware in this work, the nature of the relationship between the proposed firmware architecture and the extended Spark environment was described in section 3.3.5 of chapter 3; the extended Spark environment employs a server-side firmware instance to control the NAA compute nodes through a set of NAA commands. The extended Spark environment may define new transformations that are capable of targeting NAA compute nodes such as `mapFpga(path)` which attempt to allocated NAA compute nodes from a pool of available nodes, program their FPGAs, establish a complex compute pipeline, and execute multiple task on this pipeline.

The proposed NAA system architecture and its firmware architecture may be employed to seamlessly integrate NAA compute nodes into the data center. Moreover, deploying FPGAs using the NAA deployment model, improves scalability which is one of the most important design factors in modern data center architectures. Since extending the Spark middleware is outside the scope of this work, we only evaluated the performance of a single NAA compute node and compared it against a CPU processor core in chapter 4.

In chapter 4, we aimed at accelerating the multi-layer convolution operation which is one of the key kernel operations in Deep Convolutional Neural Networks (ConvNet) [8]. We employed a similar setting to the proposed NAA system architecture in chapter 3. We also employed ZedBoards [35], which are Zynq SoC [33] evaluation and development boards, to implement an accelerator for the

2D-Multi-Layer Convolution Operation. Previous studies have shown that 90% of the time spent computing the inference function in ConvNets is wasted on the 2D-Multi-Layer convolution operation [98]. The Multi-Layer Convolution filter that we implemented was of size 5×5 , which is the only filter size required in LeNet [99], a ConvNet used in handwritten digits recognition. We measured the computational latency of our ZedBoard implementation, which was running at 150 MHz, and compared it to the latency on a Core i7 processor core running at 2 GHz. We reported up to 10 times speedup ratios over the Core i7 processor core. Moreover, we developed a latency model on both the FPGA and processor core and we concluded that up to 134 times speedup may be achieved using only the ZedBoard platform. Theoretically, in terms of performance, a 134 times speedup means that, in certain situations, one FPGA device may replace around 134 Core i7 processor cores. In terms of energy, the board power of the ZedBoard was observed to be around 2.8 Watts; given that a Core i7 processor core consumes at least 21.25 Watts even when it is in an idle state, the maximum gain in energy efficiency was computed to be around 1017 times.

The operation, we accelerated in chapter 4, was implemented in isolation. In practical settings, however, the 2D-Multi-layer convolution is implemented as part of a larger application such as a complete ConvNet inference task. In chapter 5, we implemented a complete accelerator for Convolutional Neural Networks. We also proposed a design methodology for mapping any Convolutional Neural Network to an FPGA accelerator. The proposed methodology is based on Graphical Representations of DSP- and DataFlow- oriented tasks. We also proposed a novel multi-rate and stride aware methodology for efficiently implementing Convolution filters with strides configurations. In addition, to covering the design of 2D-convolution filters, we also managed to devise a methodology for mapping

all neural layers, that are typically employed in ConvNets, into hardware filters. Finally, we tested our accelerator on two different FPGA platforms: (1) a ZedBoard [35], and (2) a Cyclone V development kit [36]. We have shown that we can achieve a potential performance of up to 6 *frames/sec* on the ZedBoard and 9 *frames/sec* on the Cyclone V devkit. In terms of board powers consumption, the ZedBoard consumes around 2.95 Watts, whereas the Cyclone V consumes around 8.5 Watts.

The latency of a software implementation of the AlexNet inference on a Core i7 processor core (Core i7-4510U) was measured and found to be around 351 *msec* i.e., 2.84 *frames/sec*. The potential speedup ratios when the ZedBoard and the Cyclone V devkit are employed are 2.11 \times and 3.16 \times respectively. Given that a Core i7 processor core consumes at least 21.25 Watts, the Energy reduction ratios, when the ZedBoard and the Cyclone V devkit are employed, may be estimated as 15.2 \times with the ZedBoard, and 7.9 \times with the Cyclone V. Note that in this implementation, the resource utilization of all types of FPGA resources are only less than 31%. By utilizing all the available FPGA resources, a more efficient implementation may be created, and thus the speedup and energy reduction ratios may be both improved significantly; we left this optimization for a future work (refer to section 6.2.2 for more information on the potential speedup results when all resources are used)

In the following we summarize all the conclusions drawn from this work:

- FPGAs may be deployed into the data center environment as Network-Attached Accelerator (NAA) nodes. A compute model and a firmware architecture were developed to allow modified versions of cluster computing frameworks such as an extended version of Spark to target the FPGA nodes.

- Since NAA nodes are attached directly into the data center network infrastructure, an NAA-based cluster may be scaled out easily by simply increasing the number of NAA nodes.
- A modular design that provides flexibility for multi-layer ConvNet implementations with different topologies.
- A single FPGA compute node (NAA compute node) may provide orders of magnitude speedup ($134\times$) and energy reduction ($1017\times$) ratios over a Core i7 processor core, when implementing a simple operation such as the 2D-Mult-layer Convolution operation.
- A design methodology for mapping Convolutional Neural Networks (ConvNet) to low-end and resource-constrained FPGA devices was devised. We used this methodology to map AlexNet, a computationally complex ConvNet architecture, to two low-end FPGA platforms: ZedBoard, and Cyclone V devkit. Although the mapped designs utilize only less than 31% of the available FPGA resources, they have shown promising results. Employing the remaining resources would significantly improve the performance and energy results (refer to section 6.2.2).

6.2 Future Perspectives

6.2.1 Network Attached Accelerator Optimizations

In chapter 3, we described the internal structure of the NAA compute node as being made up of three essential components: (1) a general-purpose multi-core microprocessor component, (2) a reconfigurable component (FPGA), and (3) a

Networking component. The Networking component is responsible for receiving/transmitting data across the local area network. Recall that we employed a lightweight version of the TCP/IP protocol stack to allow the NAA compute node to reliably communicate with one another and with the server nodes. Since the TCP/IP protocol stack runs on the multi-core processor component of the NAA node, TCP performance or the achievable throughput on the NAA nodes is bounded by the computational capacity of the processor. As a general rule of thumb, every 1 bit/sec of TCP/IP throughput requires 1 Hertz of CPU processing [134]. For instance, to maintain a stable bi-directional 200 Mbit/sec TCP/IP connection between two NAA nodes (an aggregate throughput of 400 Mbit/sec), the processor component would have to spend 400 Million clock cycles per second for the TCP/IP stack alone. If the NAA node has a single processor core, and the processor core is clocked at 400 MHz, then there will be no more rooms left for other tasks beside the TCP/IP protocol stack. In our experimental setup, we employed ZedBoards to implement NAA nodes, mainly because ZedBoards consume significantly less power than any CPU-based computer system. Although the ZedBoard does incorporate a processor, the employed processor is a lightweight dual-core arm A9 processor core that is optimized for low-end and power sensitive applications.

The conundrum we're faced with in this situation may be summarized as follows: incorporating a low-end processor core on the NAA node is a desirable feature since it reduces power consumption; however, the reduced computational capacity of the processor significantly hinders TCP performance. To solve this problem, we may deploy soft TCP offload Engines (TCPoE) in the FPGA fabric to relieve the processor core from running the TCP/IP protocol stack. Many FPGA manufacturers and third-party IP core vendors provide production grade

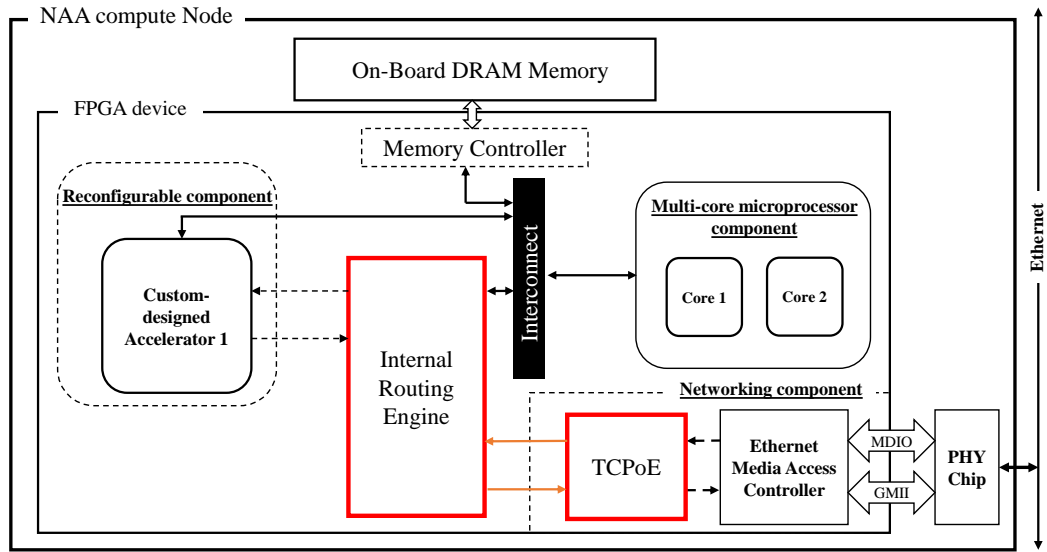


Figure 6.1: Modified NAA Node Architecture

TCPoE that may even run at wire speed [135]. Figure x shows a modified NAA node architecture that incorporates a TCPoE near the Networking component. In addition to employing a TCPoE, we may also employ an Internal Routing and Forwarding Offload Engine (IRFoE). In the current implementation of the NAA compute node, we assigned the task of routing the NAA node’s internal traffic to the processor core. As in the case of the TCP/IP protocol stack, the performance of the internal routing and forwarding function may also be bounded by the computational capacity of the processor core. Consequently, offloading the routing and forwarding roles to a custom-designed circuit significantly improves application performance. Designing, implementing, profiling and testing the IRFoE are left to a future work.

6.2.2 Optimizing FeatherNet for Performance

In chapter 5 and specifically in section 5.3.3, we have shown that a hardware realization of a 2-D convolution filter with a filter mask of size $P \times Q$, a horizontal

stride configuration s_h , a vertical stride configuration s_v , a horizontal padding configuration p_h , a vertical padding configuration p_v would result in L adders, multipliers and registers, M delay lines, and $L + M$ multiplexers, where L is defined as:

$$L(P, Q, s_h, s_v, p_h, p_v) = \left(\left\lceil \left\lceil \frac{P - p_h}{s_h} \right\rceil + \frac{p_h}{s_h} - 1 \right\rceil + 1 \right) \times \left(\left\lceil \left\lceil \frac{Q - p_v}{s_v} \right\rceil + \frac{p_v}{s_v} - 1 \right\rceil + 1 \right) \quad (6.1)$$

Moreover, when we described the 2D- coefficient delivery pattern and its corresponding Coefficient Scrambler object, we mentioned that the repetition attributes may be obtained using the following equations:

$$R = S_I + 2 \times p_h - P + s_h \quad (6.2)$$

$$R' = S'_I + 2 \times p_v - Q + s_v \quad (6.3)$$

Where S_I and S'_I are the width and height of the input image. Consequently, the actual latency for computing an output feature map of size $S_I \times S'_I$ is $R \times R'$. We can devise an architecture that utilizes all the available resources on the FPGA in such a way that when computing a single layer, the architecture attempts to compute as many convolutions in parallel as it can afford. With this method, given that there are M available multiplier on the FPGA, the total number of convolutions that may be computed in parallel (TNOCP) when computing a single ConvNet layer is:

$$TNOCP(M, P, Q, s_h, s_v, p_h, p_v) = \left\lfloor \frac{M}{L(P, Q, s_h, s_v, p_h, p_v)} \right\rfloor \quad (6.4)$$

Given a layer i , with a filter mask of size $P^i \times P^i$, horizontal and vertical stride configurations s^i , horizontal and vertical padding configurations p^i , I^i input feature maps, O^i output feature maps, a grouping parameter g^i , and input feature map size $S_I^i \times S_I^i$. Assuming that all intermediate feature maps are being cached on BRAMs, the latency of computing layer i with M multipliers may be estimated using the following equation:

$$L^i(P^i, s^i, p^i, I^i, O^i, g^i, M) = \frac{I^i \times O^i \times (S_I^i + 2 \times p^i - P^i + s^i)^2}{g^i \times TNOC P(P, Q, s_h, s_v, p_h, p_v)} \quad (6.5)$$

The latency of computing all the convolutional layers then becomes:

$$L_c(x) = T_{image} + T_{coef} + \sum_i L^i(P^i, s^i, p^i, I^i, O^i, g^i, M) \quad (6.6)$$

T_{coef} , here, is the aggregate time it takes to read all the coefficients or parameter of the convolutional layers, and T_{image} is the time it takes to read the input image. We have shown in chapter 5, that fully connected layers are memory bound, hence the latency of computing the output of fully connected layers is exactly equal to the time it takes to read all the parameters for the fully connected layers ($T_{connected}$). The total latency for computing the inference, then, becomes:

$$L_c(x) = T_{image} + T_{coef} + T_{connected} + \sum_i L^i(P^i, s^i, p^i, I^i, O^i, g^i, M) \quad (6.7)$$

On the ZedBoard every DSP unit may implement one 18×25 signed multiplier; and since there are 240 DSP unit on the Zynq 7020 device, the total number of independent multipliers is $M_{Zynq} = 240$. On the Cyclone Devkit every DSP unit may implement 2 independent 18×18 multipliers. Since there are 342 DSP

units, the total number of independent multipliers is $M_{Cyclone} = 640$. Consequently, we can estimate the maximum achievable performance on every device using the model we provided in equation 6.7. For instance, the maximum achievable performances on the Cyclone V Devkit and on the Zedboard, when they implement AlexNet, are 31 *frames/sec* and 21 *frames/sec*, respectively. Employing all the FPGA resources to compute as many convolutions in parallel as possible requires further designing and testing, and we leave this for a future work.

Appendix A

Proof for the Proposed Transformation in Figure 5.7

Section 5.3.3 presents a method for transferring delay elements from the input of an M-fold compressor to its output. Figure 5.7(b) can be considered a generalized form of the first noble identity. In the following, we will prove the validity of the transformation shown in Figure 5.7(b), knowing that the transformation in Figure 5.7(a) is valid. Assuming we have delay T elements followed by a M-fold sample rate compressor as shown in figure A.1.

The T delay elements can be rearranged into S delay element followed by $T - S$ delay elements; here S can be any natural number. The rearrangement is shown in Figure A.2.

To employ the Noble identity shown in Figure 5.7(a) on the M-fold compressor and the $T - S$ delay elements, we choose S such that the value of $T - S$ is a



Figure A.1: T delay elements followed by an M-fold compressor

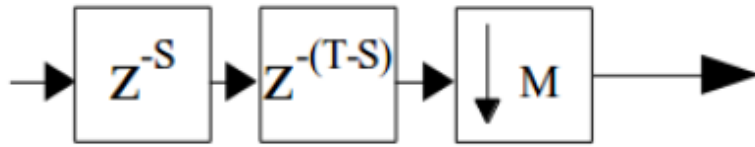


Figure A.2: T delay elements rearranged into S delay element followed by $T - S$ delay elements

multiple of M i.e., $T - S = i \times M$. In this case $S = T - i \times M$. After we employ the Noble identity in Figure 5.7(a), we end up with the transformation proposed in Figure 5.7(b).

Bibliography

- [1] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, “Big data: The next frontier for innovation, competition, and productivity,” McKinsey Global Institute, Tech. Rep., 5 2011.
- [2] C. Eaton, D. Deroos, T. Deutsch, G. Lapis, and P. Zikopoulos, *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill, 2012.
- [3] (2018) Sensor suppliers to the civil engineering industry. <http://www.sherbornesensors.com/international/civil-engineering>. Sherborne Sensors. [Online]. Available: <http://www.sherbornesensors.com/international/civil-engineering>
- [4] (2013, Mar) Ucla relies on breakthrough ‘big data’ technology from ibm to help patients with traumatic brain injuries. <https://www-03.ibm.com/press/us/en/pressrelease/40624.wss>. IBM. [Online]. Available: <https://www-03.ibm.com/press/us/en/pressrelease/40624.wss>
- [5] V. Turner, J. F. Gantz, D. Reinsel, and S. Minton, “The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things,” International Data Corporation, Tech. Rep., 4 2014.
- [6] G. Rohling. (2014, October) Facts and forecasts: Billions of things, trillions of dollars. [Online]. Available: <https://www.siemens.com/innovation/en/home/pictures-of-the-future/digitalization-and-software/internet-of-things-facts-and-forecasts.html>
- [7] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald, and E. Muharemagic, “Deep learning applications and challenges in big data analytics,” *Journal of Big Data*, vol. 2, no. 1, p. 1, Feb 2015. [Online]. Available: <https://doi.org/10.1186/s40537-014-0007-7>
- [8] M. A. Nielsen, *Neural Network and Deep Learning*. Determination Press, 2015.
- [9] A. Krizhevsky and G. E. Hinton, “Using very deep autoencoders for content-based image retrieval.” in *ESANN*, 2011.

- [10] R. Mottaghi, H. Bagherinezhad, M. Rastegari, and A. Farhadi, “Newtonian scene understanding: Unfolding the dynamics of objects in static images,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 3521–3529.
- [11] J. McCormac, A. Handa, S. Leutenegger, and A. J. Davison, “Scenet rgb-d: 5m photorealistic images of synthetic indoor trajectories with ground truth,” *arXiv preprint arXiv:1612.05079*, 2016.
- [12] Y. Lecun, Y. Benjio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, pp. 436–444, may 2015.
- [13] A. Krizhevsky, I. Sutskever, and G. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems*, Nevada, NV, 2012, pp. 1097–1105.
- [14] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: <http://doi.acm.org/10.1145/1327452.1327492>
- [15] Apache hadoop. <http://hadoop.apache.org/>. Apache. [Online]. Available: <http://hadoop.apache.org/>
- [16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks,” in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys ’07. New York, NY, USA: ACM, 2007, pp. 59–72. [Online]. Available: <http://doi.acm.org/10.1145/1272996.1273005>
- [17] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, “Large scale distributed deep networks,” in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS’12. USA: Curran Associates Inc., 2012, pp. 1223–1231. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2999134.2999271>
- [18] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1863103.1863113>
- [19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A system for large-scale graph processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’10.

- New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: <http://doi.acm.org/10.1145/1807167.1807184>
- [20] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed graphlab: A framework for machine learning and data mining in the cloud,” *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012. [Online]. Available: <https://doi.org/10.14778/2212351.2212354>
- [21] T. W. Dinsmore. (2017, February) Spark is the future of analytics. <https://thomaswdinsmore.com/2017/02/14/spark-is-the-future-of-analytics/>. ML/DL. [Online]. Available: <https://thomaswdinsmore.com/2017/02/14/spark-is-the-future-of-analytics/>
- [22] J. Gantz and D. Reinsel, “The Digital Universe Decade – Are You Ready?” International Data Corporation, Tech. Rep., 5 2010.
- [23] J. Whitney and P. Delforge, “Data Center Efficiency Assessment,” Natural Resources Defense Council (NRDC), Tech. Rep., 2014.
- [24] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-based Computation*. Burlington, MA: Morgan Kaufmann, 2008.
- [25] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing fpga-based accelerator design for deep convolutional neural networks,” in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’15. New York, NY, USA: ACM, 2015, pp. 161–170. [Online]. Available: <http://doi.acm.org/10.1145/2684746.2689060>
- [26] U. Aydonat, 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*, ser. FPGA ’17. New York, NY, USA: ACM, 2017, pp. 55–64. [Online]. Available: <http://doi.acm.org/10.1145/3020078.3021738>
- [27] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, “Neuflow: A runtime reconfigurable dataflow processor for vision,” in *CVPR 2011 WORKSHOPS*, June 2011, pp. 109–116.
- [28] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, “A reconfigurable fabric for accelerating large-scale datacenter services,” *SIGARCH Comput.*

- Archit. News*, vol. 42, no. 3, pp. 13–24, Jun. 2014. [Online]. Available: <http://doi.acm.org/10.1145/2678373.2665678>
- [29] Apache spark. <https://spark.apache.org/>. Apache. [Online]. Available: <https://spark.apache.org/>
- [30] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 1–9.
- [31] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks,” in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 818–833.
- [32] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: <http://arxiv.org/abs/1409.1556>
- [33] (2017) Zynq-7000: All programmable soc with hardware and software programmability. <https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>. Xilinx. [Online]. Available: <https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>
- [34] (2017) Cyclone v. <https://www.altera.com/products/fpga/cyclone-series/cyclone-v/overview.html>. Intel/Altera. [Online]. Available: <https://www.altera.com/products/fpga/cyclone-series/cyclone-v/overview.html>
- [35] (2017) Zedboard. <http://zedboard.org/product/zedboard>. Avnet. [Online]. Available: <http://zedboard.org/product/zedboard>
- [36] (2017) Cyclone v - gx fpga development kit. https://www.altera.com/products/boards_and_kits/dev-kits/altera/kit-cyclone-v-gx.html. Intel/Altera. [Online]. Available: https://www.altera.com/products/boards_and_kits/dev-kits/altera/kit-cyclone-v-gx.html
- [37] P. K. Gupta. (2015, June) Xeon+fpga platform for the data center. <https://www.archive.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf>. Intel. [Online]. Available: <https://www.archive.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf>
- [38] (2014) The xilinx sdaccel development environment. <https://www.xilinx.com/support/documentation/backgrounders/sdaccel-backgrounder.pdf>. Xilinx. [Online]. Available: <https://www.xilinx.com/support/documentation/backgrounders/sdaccel-backgrounder.pdf>

- [39] “Accelerating High-Performance Computing With FPGAs,” Altera Corporation, Tech. Rep., 10 2007.
- [40] (2008, May) Data center architecture overview. https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/DC_Infra2.5/DCInfra_1.html. Cisco. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/DC_Infra2.5/DCInfra_1.html
- [41] “Telecommunications Infrastructure Standard for Data Centers ANSI/TIA-942,” TELECOMMUNICATIONS INDUSTRY ASSOCIATION, Arlington, VA 22201 U.S.A., Standard, 2005.
- [42] “Data Center Design and Implementation Best Practices,” BICSI, 8610 Hidden River Parkway Tampa, FL 33637-1000 USA, Standard, 2014.
- [43] “Cost of Data Center Outages,” Ponemon Institute, Tech. Rep., January 2016.
- [44] “Amendment to carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications-aggregation of multiple link segments,” *IEEE Std 802.3ad-2000*, pp. i–173, 2000.
- [45] W. W. Eckerson, “Three tier client/server architecture: Achieving scalability, performance and efficiency in client server applications,” *Open Information Systems*, vol. 10, no. 1, 1995.
- [46] T. Plachetka, “Event-driven message passing and parallel simulation of global illumination,” Ph.D. dissertation, University of Paderborn, 2003.
- [47] J. S. Park, M.-S. Chen, and P. S. Yu, “Efficient parallel data mining for association rules,” in *Proceedings of the Fourth International Conference on Information and Knowledge Management*, ser. CIKM ’95. New York, NY, USA: ACM, 1995, pp. 31–36. [Online]. Available: <http://doi.acm.org/10.1145/221270.221320>
- [48] Q. V. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng, “Building high-level features using large scale unsupervised learning,” in *Proceedings of the 29th International Conference on International Conference on Machine Learning*, ser. ICML’12. USA: Omnipress, 2012, pp. 507–514. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3042573.3042641>
- [49] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li, “Parallelized stochastic gradient descent,” in *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS’10.

- USA: Curran Associates Inc., 2010, pp. 2595–2603. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2997046.2997185>
- [50] (2006) Fibre channel basics. https://images.apple.com/server/docs/Fibre_Channel_Basics_TB_v10.4.pdf. Apple. [Online]. Available: https://images.apple.com/server/docs/Fibre_Channel_Basics_TB_v10.4.pdf
- [51] Cloud dataflow. <https://cloud.google.com/dataflow/>. Google. [Online]. Available: <https://cloud.google.com/dataflow/>
- [52] S. J. Bigelow. (2018) Scale-up or scale-out: What fits best in your data center? <https://searchdatacenter.techtarget.com/tip/Scale-up-or-scale-out-What-fits-best-in-your-data-center>. SearchDataCenter. [Online]. Available: <https://searchdatacenter.techtarget.com/tip/Scale-up-or-scale-out-What-fits-best-in-your-data-center>
- [53] (2018, 05) Computer processor history. <https://www.computerhope.com/history/processor.htm>. Computer Hope. [Online]. Available: <https://www.computerhope.com/history/processor.htm>
- [54] N. Abbani, A. Ali, D. A. Otoom, M. Jomaa, M. Sharafeddine, H. Artail, H. Akkary, M. A. R. Saghier, M. Awad, and H. Hajj, “A distributed reconfigurable active ssd platform for data intensive applications,” in *2011 IEEE International Conference on High Performance Computing and Communications*, Sept 2011, pp. 25–34.
- [55] F. Gallagher. (2013) The big data value chain. <http://fraysen.blogspot.sg/2012/06/big-data-value-chain.html>. Apple. [Online]. Available: <http://fraysen.blogspot.sg/2012/06/big-data-value-chain.html>
- [56] H. Hu, Y. Wen, T. S. Chua, and X. Li, “Toward scalable systems for big data analytics: A technology tutorial,” *IEEE Access*, vol. 2, pp. 652–687, 2014.
- [57] (2017) Internet usage statistics the internet big picture. <https://www.internetworldstats.com/stats.htm>. Internet World Stats. [Online]. Available: <https://www.internetworldstats.com/stats.htm>
- [58] T. Hale. (2017, July) How much data does the world generate every minute? <http://www.iflscience.com/technology/how-much-data-does-the-world-generate-every-minute/>. iflscience. [Online]. Available: <http://www.iflscience.com/technology/how-much-data-does-the-world-generate-every-minute/>
- [59] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, “Helios: A hybrid

- electrical/optical switch architecture for modular data centers,” in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 339–350. [Online]. Available: <http://doi.acm.org/10.1145/1851182.1851223>
- [60] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center tcp (dctcp),” in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: <http://doi.acm.org/10.1145/1851182.1851192>
- [61] B. Vamanan, J. Hasan, and T. Vijaykumar, “Deadline-aware datacenter tcp (d2tcp),” in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 115–126. [Online]. Available: <http://doi.acm.org/10.1145/2342356.2342388>
- [62] P. Francisco. (2014) Ibm puredata system for analytics architecture. <http://www.redbooks.ibm.com/redpapers/pdfs/redp4725.pdf>. IBM's International Technical Support Organization. [Online]. Available: <http://www.redbooks.ibm.com/redpapers/pdfs/redp4725.pdf>
- [63] S. Ghemawat, H. Gombioff, and S.-T. Leung, “The google file system,” in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43. [Online]. Available: <http://doi.acm.org/10.1145/945445.945450>
- [64] Apache hadoop yarn. <https://hortonworks.com/apache/yarn/>. Apache. [Online]. Available: <https://hortonworks.com/apache/yarn/>
- [65] Containers at google. <https://cloud.google.com/containers/>. Google. [Online]. Available: <https://cloud.google.com/containers/>
- [66] Mahout. <https://mahout.apache.org/>. Apache. [Online]. Available: <https://mahout.apache.org/>
- [67] Apache hive tm. <https://hive.apache.org/>. Apache. [Online]. Available: <https://hive.apache.org/>
- [68] Manage massive amounts of data, fast, without losing sleep. <http://cassandra.apache.org/>. Apache. [Online]. Available: <http://cassandra.apache.org/>
- [69] M. Zaharia, *An Architecture for Fast and General Data Processing on Large Clusters*. USA: ACM Books, 2016.

- [70] Apache mesos. <http://mesos.apache.org/>. Apache. [Online]. Available: <http://mesos.apache.org/>
- [71] Production-grade container orchestration. <https://kubernetes.io/>. Kubernetes. [Online]. Available: <https://kubernetes.io/>
- [72] Rdd programming guide. <https://spark.apache.org/docs/latest/rdd-programming-guide.html>. Apache Spark. [Online]. Available: <https://spark.apache.org/docs/latest/rdd-programming-guide.html>
- [73] “INTEL CORP, FORM 10-K,” EDGAR Online, Inc., Tech. Rep., 12 2016.
- [74] D. Strigl, K. Kofler, and S. Podlipnig, “Performance and scalability of gpu-based convolutional neural networks,” in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Feb 2010, pp. 317–324.
- [75] Mapreduce tutorial. <http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>. Apache. [Online]. Available: <http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>
- [76] (2015, Dec) Intel completes acquisition of altera. <https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/>. Intel. [Online]. Available: <https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/>
- [77] T. P. Morgan. (2018, May) A peek inside that intel xeon-fpga hybrid chip. <https://www.nextplatform.com/2018/05/24/a-peek-inside-that-intel-xeon-fpga-hybrid-chip/>. The Next Platform. [Online]. Available: <https://www.nextplatform.com/2018/05/24/a-peek-inside-that-intel-xeon-fpga-hybrid-chip/>
- [78] N. Abbani, A. Ali, D. A. Otoom, M. Jomaa, M. Sharafeddine, H. Artail, H. Akkary, M. A. R. Saghir, M. Awad, and H. Hajj, “A distributed reconfigurable active ssd platform for data intensive applications,” in *2011 IEEE International Conference on High Performance Computing and Communications*, Sept 2011, pp. 25–34.
- [79] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, “Design tradeoffs for ssd performance,” in *USENIX 2008 Annual Technical Conference*, ser. ATC’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 57–70. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1404014.1404019>

- [80] M. Jomaa, K. Mershad, N. Abbani, Y. Sharaf-Dabbagh, B. Romanous, H. Artail, M. A. R. Saghir, H. Hajj, H. Akkary, and M. Awad, “A mediation layer for connecting data-intensive applications to reconfigurable data nodes,” in *2013 22nd International Conference on Computer Communication and Networks (ICCCN)*, July 2013, pp. 1–9.
- [81] A. Ali, M. A. R. Saghir, H. Akkary, H. Artail, H. Hajj, and M. Awad, “Rassd: A dynamically reconfigurable active storage device for energy efficient data analytics,” in *2013 4th Annual International Conference on Energy Aware Computing Systems and Applications (ICEAC)*, Dec 2013, pp. 81–86.
- [82] A. Kaitoua, H. Hajj, M. A. R. Saghir, H. Artail, H. Akkary, M. Awad, M. Sharafeddine, and K. Mershad, “Hadoop extensions for distributed computing on reconfigurable active ssd clusters,” *ACM Trans. Archit. Code Optim.*, vol. 11, no. 2, pp. 22:1–22:26, Jul. 2014. [Online]. Available: <http://doi.acm.org/10.1145/2608199>
- [83] A. Ali, M. Jomaa, B. Romanous, M. Sharafeddine, M. A. R. Saghir, H. Akkary, H. Artail, M. Awad, and H. Hajj, “An operating system for a reconfigurable active ssd processing node,” in *2012 19th International Conference on Telecommunications (ICT)*, April 2012, pp. 1–6.
- [84] B. Bhattacharya and S. S. Bhattacharyya, “Parameterized dataflow modeling for dsp systems,” *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, Oct 2001.
- [85] K. Desnos, M. Pelcat, J. Nezan, S. S. Bhattacharyya, and S. Aridhi, “Pimm: Parameterized and interfaced dataflow meta-model for mpsoes runtime reconfiguration,” in *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, July 2013, pp. 41–48.
- [86] (2015) lwip: A lightweight tcp/ip stack. <https://savannah.nongnu.org/projects/lwip/>. Free Software Foundation, Inc. [Online]. Available: <https://savannah.nongnu.org/projects/lwip/>
- [87] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “Imagenet large scale visual recognition challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, Dec 2015. [Online]. Available: <https://doi.org/10.1007/s11263-015-0816-y>
- [88] (2016) Imagenet. <http://image-net.org/>. Stanford Vision Lab. [Online]. Available: <http://image-net.org/>

- [89] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, “Cnp: An fpga-based processor for convolutional networks,” in *2009 International Conference on Field Programmable Logic and Applications*, Aug 2009, pp. 32–37.
- [90] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, “Hardware accelerated convolutional neural networks for synthetic vision systems,” in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, May 2010, pp. 257–260.
- [91] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, “A 240 gops/s mobile coprocessor for deep neural networks,” in *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, June 2014, pp. 696–701.
- [92] Y. Jia, 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*, ser. MM ’14. New York, NY, USA: ACM, 2014, pp. 675–678. [Online]. Available: <http://doi.acm.org/10.1145/2647868.2654889>
- [93] Caffe. <http://caffe.berkeleyvision.org/>. Berkeley Artificial Intelligence Research. [Online]. Available: <http://caffe.berkeleyvision.org/>
- [94] Torch: A scientific computing framework for luajit. <http://torch.ch/>. Torch. [Online]. Available: <http://torch.ch/>
- [95] Theano. <http://deeplearning.net/software/theano/>. Theano. [Online]. Available: <http://deeplearning.net/software/theano/>
- [96] Tensorflow: An open source machine learning framework for everyone. <https://www.tensorflow.org/>. Google. [Online]. Available: <https://www.tensorflow.org/>
- [97] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML’15. JMLR.org, 2015, pp. 1737–1746. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3045118.3045303>
- [98] A. Alhamali, N. Salha, R. Morcel, M. Ezzeddine, O. Hamdan, H. Akkary, and H. Hajj, “Fpga-accelerated hadoop cluster for deep learning computations,” in *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, Nov 2015, pp. 565–574.

- [99] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.
- [100] Cifar. <https://www.cs.toronto.edu/~kriz/cifar.html>. CIFAR. [Online]. Available: <https://www.cs.toronto.edu/~kriz/cifar.html>
- [101] D. Blaauw, D. Sylvester, P. Dutta, Y. Lee, I. Lee, S. Bang, Y. Kim, G. Kim, P. Pannuto, Y. . Kuo, D. Yoon, W. Jung, Z. Foo, Y. . Chen, S. Oh, S. Jeong, and M. Choi, “Iot design space challenges: Circuits and systems,” in *2014 Symposium on VLSI Technology (VLSI-Technology): Digest of Technical Papers*, June 2014, pp. 1–2.
- [102] H. Jayakumar, K. Lee, W. S. Lee, A. Raha, Y. Kim, and V. Raghunathan, “Powering the internet of things,” in *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, Aug 2014, pp. 375–380.
- [103] H. Singh. (2018, May) How much does it cost to develop an iot application? [Online]. Available: <http://customerthink.com/how-much-does-it-cost-to-develop-an-iot-application/>
- [104] P. K. Keshab, *VLSI Digital Signal Processing Systems Design and Implementation*. New York, NY: Wiley & Sons, 1999.
- [105] AlphaData, “An open source fpga cnn library,” 2017. [Online]. Available: ftp://ftp.alpha-data.com/pub/appnotes/cnn/ad-an-0055.v1_0.pdf
- [106] K. Abdelouahab, M. Pelcat, J. Sérot, C. Bourrasset, and F. Berry, “Tactics to directly map cnn graphs on embedded fpgas,” *IEEE Embedded Systems Letters*, vol. 9, no. 4, pp. 113–116, Dec 2017.
- [107] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, “From high-level deep neural models to fpgas,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.
- [108] P. K. Meher, S. Chandrasekaran, and A. Amira, “Fpga realization of fir filters by efficient and flexible systolization using distributed arithmetic,” *IEEE Transactions on Signal Processing*, vol. 56, no. 7, pp. 3009–3017, July 2008.
- [109] X. Liang, J. Jean, and K. Tomko, “Data buffering and allocation in mapping generalized template matching on reconfigurable systems,” *J. Supercomput.*, vol. 19, no. 1, pp. 77–91, May 2001. [Online]. Available: <https://doi.org/10.1023/A:1011196613858>

- [110] F. Cardells-Tormo, P. . Molinet, J. Sempere-Agullo, L. Baldez, and M. Bautista-Palacios, “Area-efficient 2d shift-variant convolvers for fpga-based digital image processing,” in *International Conference on Field Programmable Logic and Applications, 2005.*, Aug 2005, pp. 578–581.
- [111] H. Zhang, M. Xia, and G. Hu, “A multiwindow partial buffering scheme for fpga-based 2-d convolvers,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 2, pp. 200–204, Feb 2007.
- [112] C. D. Moreno, F. J. Quiles, M. A. Ortiz, M. Brox, J. Hormigo, J. Villalba, and E. L. Zapata, “Efficient mapping on fpga of convolution computation based on combined csa-cpa accumulator,” in *2009 16th IEEE International Conference on Electronics, Circuits and Systems - (ICECS 2009)*, Dec 2009, pp. 419–422.
- [113] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: <http://doi.acm.org/10.1145/1498765.1498785>
- [114] (2017) Vivado high level synthesis. <https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>. Xilinx. [Online]. Available: <https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>
- [115] (2017) Intel sdk for opencl applications. <https://software.intel.com/en-us/intel-opencl>. Intel. [Online]. Available: <https://software.intel.com/en-us/intel-opencl>
- [116] M. Alwani, H. Chen, M. Ferdman, and P. Milder, “Fused-layer cnn accelerators,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.
- [117] Y. Shen, M. Ferdman, and P. Milder, “Maximizing cnn accelerator efficiency through resource partitioning,” in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 535–547.
- [118] D. G. Bailey, “The advantages and limitations of high level synthesis for fpga based image processing,” in *Proceedings of the 9th International Conference on Distributed Smart Cameras*, ser. ICDSC ’15. New York, NY, USA: ACM, 2015, pp. 134–139. [Online]. Available: <http://doi.acm.org/10.1145/2789116.2789145>

- [119] P. Peng, Y. Mingyu, and X. Weisheng, “Running 8-bit dynamic fixed-point convolutional neural network on low-cost arm platforms,” in *2017 Chinese Automation Congress (CAC)*, Oct 2017, pp. 4564–4568.
- [120] V. Vanhoucke, A. Senior, and M. Z. Mao, “Improving the speed of neural networks on cpus,” in *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, vol. 1, 2011, p. 4.
- [121] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, “Fixed point quantization of deep convolutional networks,” in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML’16. JMLR.org, 2016, pp. 2849–2858. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3045390.3045690>
- [122] P. Gysel, “Ristretto: Hardware-oriented approximation of convolutional neural networks,” *arXiv preprint arXiv:1605.06402*, 2016.
- [123] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1,” *arXiv preprint arXiv:1602.02830*, 2016.
- [124] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” *arXiv preprint arXiv:1510.00149*, 2015.
- [125] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Quantized neural networks: Training neural networks with low precision weights and activations,” *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6869–6898, Jan. 2017. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3122009.3242044>
- [126] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.
- [127] F. Li and B. Liu, “Ternary weight networks,” *CoRR*, vol. abs/1605.04711, 2016. [Online]. Available: <http://arxiv.org/abs/1605.04711>
- [128] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “Xnor-net: Imagenet classification using binary convolutional neural networks,” in *European Conference on Computer Vision*. Springer, 2016, pp. 525–542.
- [129] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, “Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients,” *arXiv preprint arXiv:1606.06160*, 2016.

- [130] D. E. Dudgeon and R. M. Mersereau, *Multidimensional Digital Signal Processing*, ser. Prentice-Hall Signal Processing Series, Englewood Cliffs, NJ, 1984.
- [131] A. V. Oppenheim, R. W. Schaffer, and J. R. Buck, *Discrete-Time Signal Processing*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1999.
- [132] (2015) Model zoo. <https://github.com/BVLC/caffe/wiki/Model-Zoo>. BVLC. [Online]. Available: <https://github.com/BVLC/caffe/wiki/Model-Zoo>
- [133] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, “An introduction to high-level synthesis,” *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 8–17, July 2009.
- [134] A. P. Foong, T. R. Huff, H. H. Hum, J. R. Patwardhan, and G. J. Reginier, “Tcp performance re-visited,” in *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*. IEEE, 2003, pp. 70–79.
- [135] (2017) 10g tcp offload engine. <https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/algo-logic/ip/10g-tcp-offload-engine.html>. Intel/Altera. [Online]. Available: <https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/algo-logic/ip/10g-tcp-offload-engine.html>