Scaling Accelerator-Rich Systems for Big-Data Analytics

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Di Wu

2017
ABSTRACT OF THE DISSERTATION

Scaling Accelerator-Rich Systems for Big-Data Analytics

by

Di Wu

Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2017

Professor Jingsheng Jason Cong, Chair

As the Dennard scaling is coming to an end, the energy-density of computing devices can no longer increase. As a result, both industry and academia must seek drastic measures to sustain the scalability of computation in order to meet the ever-growing demands. Accelerators have been shown to provide orders-of-magnitude energy-efficiency and performance improvement for a number of domains. However, the complexity of accelerator programming and deployment has limited the adoption of accelerators at a larger scale. Focusing on that issue, this thesis discusses design methodologies to scale from simple single-node accelerator systems to large-scale accelerator-rich systems. In general, there are two dimensions of scaling: scaling-up and scaling-out. Scaling-up represents the scheme that partitions a complex application on to multiple accelerator nodes to exploit task-level parallelism. Scaling-out represents the scheme that replicates the same accelerator on multiple nodes for data-parallel executions. In practice, combinations of these two scaling methodologies can be explored regarding application characteristics. This thesis discusses several techniques from both scaling dimensions with case studies of the domain of big-data analytics.
The dissertation of Di Wu is approved.

Yingnian Wu
Glenn Reinman
Tyson Condie
Jingsheng Jason Cong, Committee Chair

University of California, Los Angeles

2017
To my wife Muyan for her love and support,
and to my son Chengzhou for the happiness he brought to me.
To my family and all the people who helped me throughout all these years.
# TABLE OF CONTENTS

1 **Introduction** .......................................................... 1

2 **Background and Related Work** .................................... 5
   2.1 Big-Data Analytic Frameworks ................................. 5
      2.1.1 Apache Spark ........................................ 5
      2.1.2 Apache YARN ........................................ 6
   2.2 Big-data Analytic Applications ............................... 7
      2.2.1 Distributed Machine Learning Algorithms ............ 7
      2.2.2 Spiking Neural Networks for Neuron Simulation ...... 9
      2.2.3 Convolutional Neural Networks for Image Recognition 10

3 **Application-Specific Accelerator System Design** ............. 13
   3.1 Overview .................................................... 13
   3.2 FPGA Simulation Engine for Spiking Neurons ............... 14
      3.2.1 Neural Microcircuits Modeling ....................... 15
      3.2.2 Platform-Based FPGA Implementation ............... 19
      3.2.3 Experimental Results ................................ 26
   3.3 Multi-FPGA Acceleration Platform for Convolutional Neural Networks ... 30
      3.3.1 Deeply Pipelined FPGA Cluster and Prototype System Design ... 32
      3.3.2 Multi-FPGA Design Space Exploration ............... 36
      3.3.3 Experimental Result .................................. 40
   3.4 Conclusions ................................................ 45

4 **Runtime System for Large-Scale Accelerator Deployment** ...... 46
5.6.3 K-Flow Throughput on CPU-FPGA ........................................ 100

5.7 Discussions ............................................................................... 103

5.8 Conclusions ................................................................................ 106

6 Concluding Remarks .................................................................... 107

References ....................................................................................... 109
LIST OF FIGURES

2.1 Example YARN architecture showing a client submitting jobs to the global re-
source manager. ................................................................. 6

2.2 A two-layer artificial neural network topology used in data classification . . . . . 8

2.3 Circuit architecture. A) The model circuit consists of three sheets of grid cells,
reciprocally connected with three VCO rings that receive driving inputs that en-
code movement velocity signals. The terms AMPA, NMDA, GABA indicates
the types of synapses through which neurons are connected. B) Each VCO ring
is composed of two circular layers of neurons, one excitatory and the other in-
hibitory, interconnected with one another by rotationally asymmetric weights so
that an activity bump circulates around the ring in the counterclockwise direc-
tion. C) Spike rasters show propagation of the activity bump through the VCO
layers (top two graphs) and the grid cell sheets (bottom graph) during a 3-second
simulation of a rat running at constant speed along a linear path. ............... 9

2.4 Feed-forward of a CNN model ........................................... 11

3.1 Circuit model of IAF neuron. ............................................... 16

3.2 The mapping of the neural microcircuit to our platform. The left side of the
figure illustrates the hierarchies of the neural microcircuit and the examples of
each level in the oscillatory model. The right side illustrates the corresponding
implementation in our platform-based approach. .............................. 19

3.3 Template for computing engine of a neuron population ..................... 22

3.4 Logical organization of FPGA cluster .................................. 33

3.5 The prototype system for the scale-up design ................................ 35

3.6 Design space exploration of throughput (GOPS/s) and energy efficiency (GOP-
S/J) and latency (ms) maximization for AlexNet and VGG-16 .................. 41
3.7 Execution time per image and FPGA board assignment of each layer in (a) design C and (b) design A ........................................... 42

3.8 Performance comparison between CONV and LRN with different resource budgets. The horizontal axis 'r' denotes the ratio of DSP resource for CONV and LRN; higher 'r' means more DSPs are used for convolution and less for LRN. The vertical axis denotes execution cycles. ........................................... 43

4.1 Experimental cluster with standard server node integrated with PCI-E based FPGA board from AlphaData ........................................... 48

4.2 The prototype system design .................................................. 49

4.3 Execution time (above) and energy consumption (below) normalized to the results on one Xeon server. ........................................... 53

4.4 Overview of Blaze runtime system. .......................................... 58

4.5 Node accelerator manager design to enable FPGA accelerators as a service (FaaS). 63

4.6 Single-node system performance and energy gains for each individual application. 71

4.7 Performance of LR and KM on multiple nodes. The X-axis represents the experiment configurations. For example, "4N×12T CPU" represents the configuration of 4 CPU-only nodes with 12 threads on each node. ...................... 72

4.8 Execution time breakdown for LR and KM before and after FPGA acceleration on multiple nodes. ............................... 73

4.9 Faas overhead analysis in COMP application. ............................ 74

4.10 Breakdown of the JVM-to-FPGA communication optimizations in FaaS. ..... 74

4.11 Accelerator utilization results of running a single LR application on an FPGA. 75

5.1 Baseline CPU program with multiple iterations for all N data partitions: The shadow segments indicate the part of the program that can be accelerated. In this case, both the white and shadow segments run on CPU. Throughput $T = \frac{P}{T}$. 79
5.2 Straightforward integration with fast FPGA: White segments on CPU and shadow segments on FPGA. The runtime of all \( P - 1 \) shadow segments is not longer than that of 1 white segment. FPGA speedup \( S = \frac{r(P-1)}{1-r} \). Throughput \( T = \frac{P}{S} + \frac{(1-r)t}{r} \).

Problem: CPU is idle during FPGA execution for shadow segments.

5.3 Straightforward integration with slow FPGA: White segments on CPU and shadow segments on FPGA. The runtime of all \( P - 1 \) shadow segments is longer than that of 1 white segment. In this case FPGA becomes the bottleneck when the speedup \( S = \frac{r(P-1)}{1-r} \). Throughput \( T = \frac{S}{r} \). Problem: CPU is idle during FPGA execution and FPGA becomes the bottleneck. It can be even worse than the case in 5.1 when \( S > r \cdot P \).

5.4 Theoretically optimal integration with fast FPGA: White segments on CPU and shadow segments on FPGA. This corresponds to the case in 5.2, and we omit the optimal case for 5.3. Throughput \( T = \frac{P}{(1-r)t} \).

5.5 Example K-Flow program DAGs.

5.6 Implementation of an example K-Flow program.

5.7 Primitive function/task types in K-Flow.

5.8 Overview of K-Flow dynamic scheduling system.

5.9 Throughput speedups of straightforward integration and theoretical throughput of K-Flow with \( P = 12 \) and different acceleratable ratios \( r \), compared to the CPU baseline. The X-axis is the FPGA accelerator speedup \( S \) compared to a single CPU core.

5.10 DAGs for BWA-MEM and BWA-Flow.

5.11 Throughput improvement of CPU-only BWA-Flow compared to the original BWA-MEM software, under different numbers of CPU threads (P).
5.12 Visualization of thread assignment/execution for one sample run of CPU-only BWA-Flow. The total number of CPU threads $P$ is set to 16. Each row in the figure represents the task assignment over time (seconds in X-axis) of an individual thread.

5.13 Throughput improvement over CPU baseline for various FPGA integrations: straightforward integration (Straight), BWA-Flow (K-Flow) and theoretical optimal (Theoretical).

5.14 Visualization of thread assignment/execution for one sample run of FPGA-integrated BWA-Flow. The total number of CPU threads $P$ is set to 16. Each row in the figure represents the task assignment over time (seconds in X-axis) of an individual CPU thread, except for thread 15 that represents the FPGA thread.

5.15 Example of the straightforward integration of single fast PE (top) and multiple slow PEs (bottom). The lighter shade represents CPU execution and the darker shade represents FPGA execution. The single PE on the top figure is $4\times$ faster than each of the four PEs in the bottom figure. The numbers in the labels represent parallel data partitions, and the letter 'a' and 'b' stands for CPU portion and FPGA portion of the data processing respectively. In the single-PE case, the different FPGA tasks need to be sequentialized. In the multi-PE case, four tasks can be executed in parallel. The examples show that the overall throughput of the single-PE implementation is larger because the CPU threads spend less time waiting for the FPGA execution.

5.16 Illustration of optimal overall program speedup in relation to relative FPGA kernel speedup $k = \frac{S}{P}$ and kernel ratio $r$. The optimal $k$-$r$ curve is the smallest $s$ and $r$ value to achieve a certain speedup.
LIST OF TABLES

3.1 Performance of one-second simulation of a single VCO Ring . . . . . . . . . . . . 27
3.2 Resource usage of FPGA Implementations of one VCO Ring . . . . . . . . . . . . 27
3.3 Performance of one-second-simulation of the Grid Module . . . . . . . . . . . . 28
3.4 Resource usage of FPGA Implementations of the Grid Module . . . . . . . . . . . 29
3.5 Energy Estimation of Simulating Grid Module on Different Platforms . . . . . . 30
3.6 Computation and memory complexity of different CNN layers in AlexNet [KSH12] 31
3.7 CNN Performance and energy results of different platforms . . . . . . . . . . . . 36
3.8 Comparison of CPU, GPU, FPGA implementations . . . . . . . . . . . . . . . . 44

4.1 Prototype System Specifications . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.2 Experimental Platform Configurations . . . . . . . . . . . . . . . . . . . . . . . 50
4.3 Specifications of our FPGA accelerators . . . . . . . . . . . . . . . . . . . . . . 52
4.4 Summary of experimental results on different platform configurations . . . . . 54
4.5 FPGA accelerator performance profile . . . . . . . . . . . . . . . . . . . . . . . 68
4.6 Comparison of accelerator deployment efforts in terms of lines-of-code (LOC) changes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

5.1 Application Profile of BWA for different Datasets and Platforms . . . . . . . . 101
I would like to express my sincerest gratitude to my advisor, Professor Jingsheng Jason Cong, for his continuous support of my Ph.D. studies, and for the motivation, knowledge, and guidance he gave to me. He taught me to pursue perfection, and to base my research on solving real-world problems. His lessons will continue to help me throughout the rest of my career and my life. I could not imagine having a better advisor and mentor.

Additionally, I want to thank the other members of my thesis committee: Prof. Glenn Reinman, Prof. Tyson Condie, and Prof. Yingnian Wu, for their insightful comments and encouragement, and also for the hard questions that motivated me to widen my research from various perspectives. A special thank-you goes to Prof. Tad Blair. Although he is not on my committee, he helped me a great deal in the beginning of my Ph.D. program and guided me to my first publication.

My sincere thanks also to Dr. Debbie Marr and Dr. Nick Carter, who both mentored me during my internship at Intel Labs. The scale-out accelerator design aspect in this thesis was rooted in that internship. I want to thank Dr. Peichen Pan and Dr. Peng Zhang for their collaboration during my internship at Falcon Computing Solutions, Inc., where I had the opportunity to apply my Ph.D. research to solving real industry problems.

I would also like to thank Janice Wheeler for her help in editing the language of my papers and this dissertation. She is one of the best teachers I’ve had in academic writing and the English language.

Furthermore, I want to thank the following researchers who helped and contributed to this dissertation. Chapter 3.3 is a collaboration between me and Chen Zhang. We had many discussions on the problem formulation, and Chen completed the HLS design of each individual CNN layer. Chapter 4 is a multi-year project that includes the efforts of Muhuan Huang, Cody Hao Yu, Bingjun Xiao, Hui Huang and Matteo Interlandi. Also, my thanks to Dr. Zhenman Fang for his help throughout the writing of this thesis.

Last, but not least, I am thankful to all my colleges and friends in the UCLA CDSC and
VAST lab, who fought and played together with me during all these years. I would not have survived without them.

The work in this thesis is partially supported by the NSF Expeditions in Computing (Award CCF-0926127) and CFAR (Center for Future Architecture Research, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MACRO and DARPA), and the NSF Innovation Transition (InTrans) Program (Award CCF-1436827) with Intel’s support (Award 20134321).
VITA

2007–2011  B.S., Department of Electrical Engineering,  
Tsinghua University, Beijing, China

2011–2017  Research Assistant, Computer Science Department,  
University of California, Los Angeles, USA

PUBLICATIONS

Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie,  
and Jason Cong, Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale, Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC 2016), Santa Clara, CA, October 5-7, 2016.


Young-kyu Choi, Jason Cong, and Di Wu, FPGA Implementation of EM Algorithm for 3D CT Reconstruction, Proceedings of the 22nd IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM 2014), (pp. 157-160). IEEE.

Hugh T. Blair, Allan Wu, and Jason Cong, Oscillatory neurocomputing with ring attractors: a network architecture for mapping locations in space onto patterns of neural synchrony, Philosophical Transactions of the Royal Society B: Biological Sciences 369.1635 (2014)
CHAPTER 1

Introduction

Today, computation has evolved to an unprecedented scale. A growing amount of data are being collected from web pages, social networks or online videos, while at the same time more sophisticated algorithms are being deployed in emerging applications such as customized advertisements, user behavior mining and visual/audio recognition. Service providers such as Google, Microsoft and Amazon are expanding their datacenter infrastructures to meet the demands. However, the semiconductor technology is reaching its physical limits of scaling [EBS11], which means that the power density of a single chip is no longer able to increase, and energy constraints have become the dominant limiting factors of modern datacenters. To sustain the scalability, datacenters must seek drastic methods to reduce the energy consumption for their workloads.

Many recent studies have identified that emerging big-data workloads expose scale-out characteristics [FAK12], for which the modern processor designs are over-provisioning [HQW10]. Recognizing such a mismatch between the architecture and workloads, researchers have proposed to use simple, low-power CPUs as major components in datacenters [JLC10, LGF12, AFK09]. Many large companies have also been actively engaged in research focused on this thrust. However, low-power CPUs may have larger slow-downs than power-reductions for many computationally intensive tasks [KRD12], which means that the overall energy-efficiency of them may not be better than high-end CPUs.

Customized accelerators emerge as a disruptive technology for this problem: they can match and even surpass the performance of modern server-class CPUs through specialized hardware design, without wasting energy on instruction decoding, branch misprediction, and unnecessary data movements. Although specialized hardware accelerators have a long history
and countless applications, due to the fast-changing nature of most big-data analytic applications, reconfigurable accelerators based on field-programmable gate arrays (FPGAs) have been favored by both academia and industry researchers compared to application-specific integrated circuits (ASICs). Initial prototypes of FPGA accelerators have demonstrated orders-of-magnitude performance and energy-efficiency improvement in many big-data analytic domains [CDL, CDS14, LMS13, CSJ10a]. In recent years, FPGA has become one of the mainstream acceleration platforms with the announcement of FPGA support on many public cloud providers such Amazon Web Service [ama], Alibaba Cloud [ali], Huawei Cloud [hwc] and Tencent Cloud [ten].

However, FPGAs have yet to be widely adopted by real applications at datacenter scale due to their complexity in both programming and deployment. Such complexities can be summarized into two dimensions: task dimension and data dimension.

Many modern big-data analytic applications like deep neural networks [KSH12] or web search engines [PCC14] have increasing complexity, and the entire application pipeline cannot be mapped to a single-node accelerator despite the scaling of physical devices. Some trade-off solutions include mapping partial compute kernels only, and dynamically reprogramming the node or reducing the performance so that the entire application fits on one node while matching the throughput by replicating the pipeline. Either solution will cost additional data movement, which in turn, increases the energy-consumption.

On the other end of the scale, many datacenter workloads involve rather simple computation, but a massive amount of data. A straightforward solution is to replicate the same accelerator onto many nodes and distribute data evenly across them. This solution may sound reasonable theoretically, but in practice, there are still many unsolved problems. First, in a large datacenter there will be a heterogeneous system that contains many different types of accelerators. A program should not be burdened by the complexity of scheduling tasks across nodes. Second, the usual method of utilizing an accelerator is by off-loading certain computation kernels, which involves additional data movement and other overheads. For single-node accelerator systems, the programmer can leverage techniques such as ping-pong buffering to cover such overheads. Yet for a large system, the complexity of such optimiza-
tion techniques may grow exponentially. As a result, an existing programming model does not scale well in big-data analytics.

Regarding these two dimensions of problems, this thesis proposes several methods that improve the scalability of accelerator-rich systems for big-data analytics. Just like the problems, the solutions can be summarized into two dimensions: scale-up and scale-out.

Scale-up accelerator design represents the techniques that effectively map a complex application pipeline onto multiple accelerator nodes, where each node is part of the entire pipeline. Examples of scale-up accelerator system design include recent work such as the Catapult system from Microsoft Research [PCC14] and BlueDBM from MIT [JLL15]. Both projects demonstrated a system design involving dedicated accelerator networks based on high-speed serial links that are separated from the host network. Such an accelerator network effectively virtualizes multiple nodes as a single big accelerator. These systems present a viable solution for accelerator systems that are scalable to application size; however, the decisions on how to partition the application onto different nodes still remains a manual process. Moreover, managing intra- and inter-node communications also involves great complexity. Chapter 3 of this thesis presents a methodology for generating efficient single-node and multi-node accelerator systems from high-level domain-specific languages. This methodology is demonstrated with two large-scale models in big-data analytics: spiking neural networks (SNN) and convolutional neural networks (CNN). The SNN models described in Section 3.2 are used in simulating the network of the navigational system in the rodent brain. An implementation that synthesizes efficient simulation engines from an XML description is presented. The CNN models in Section 3.3 are used for image detection and classification of the ImageNet dataset. An implementation that maps CNN representations based on Caffe [JSD14] to a multi-FPGA cluster is described. Both implementations extract basic building blocks of the application from the domain-specific languages and leverage FPGA high-level synthesis (HLS) for design space exploration to optimize performance and energy-efficiency.

Scale-out accelerator system design focuses on how to effectively deploy single-node accelerators to data-center scale. Chapter 4 first introduces different system design options with
evaluations based on prototyping systems with real-world big-data applications. From the discussion, one challenge emerges—that of how to efficiently deploy accelerators in a large-scale cluster. To address such a challenge, this thesis proposes a runtime system called Blaze that incorporates the Spark framework, an in-memory big-data analytics compute engine, with node-level accelerator managers. Blaze is designed to provide FPGA-As-A-Service (FaaS) with a friendly user interface with transparent task scheduling, and optimization schemes to reduce data movement overheads.

These two design methodologies are not mutually exclusive. In fact, the general design of accelerator-rich datacenters should include a combination of both methodologies, because big-data applications involve both complex computation and a large amount of data. Chapter 5 incorporates both the scale-up and scale-out design methodologies and introduces a job scheduling framework called K-Flow as an extension to Blaze. K-Flow is based on a dataflow programming model, and it maximizes the throughput of CPU-FPGA co-execution with a dynamic scheduler.
CHAPTER 2

Background and Related Work

Before discussing the proposed frameworks and systems, in this chapter we first provide some background information to introduce the context of this thesis.

2.1 Big-Data Analytic Frameworks

2.1.1 Apache Spark

Apache Spark [ZCF10] is a widely used fast and general large-scale data processing framework. It exposes a programming model based on Resilient Distributed Datasets (RDDs) [ZCD12]. The RDD abstraction provides transformations (e.g., map, reduce, filter, join, etc.) and actions (e.g., count, collect) that operate on datasets partitioned over a cluster of nodes. A typical Spark program executes a series of transformations ending with an action that returns a singleton value (e.g., the record count of an RDD) to the Spark driver program, which could then trigger another series of RDD transformations.

Spark caches reused data blocks in memory, often achieving significant performance speedup over the Hadoop MapReduce [had] on iterative applications such as machine learning. Recent studies [ORR15, tun] show that Spark applications are often computation-bound instead of IO or network bound in conventional Hadoop applications. This motivates us to leverage FPGAs to further accelerate the computation.

Spark can be run standalone on a cluster, or with a resource manager like Hadoop YARN [VMD13]. For each Spark application submitted to the YARN cluster, a set of containers (see Section 2.1.2) is gathered from the resource manager matching the available
resources and the application configuration. For each acquired container, the Spark context launches an *executor*: a JVM instance providing the base runtime for the execution of the actual data-processing computation (i.e., tasks), and managing the application data.

### 2.1.2 Apache YARN

YARN (Yet Another Resource Negotiator) is a widely used cluster resource management layer in the Hadoop system that allocates resources, such as CPU and memory, to multiple big-data applications (or jobs). Figure 2.1 shows a high-level view of the YARN architecture. A typical YARN setup would include a single resource manager (RM) and several node manager (NM) installations. Each NM typically manages the resources of a single machine, and periodically reports to the RM, which collects all NM reports and formulates a global view of the cluster resources. The periodic NM reports also provide a basis for monitoring the overall cluster health at the RM, which notifies relevant applications when failures occur.

A YARN job is represented by an application master (AM), which is responsible for orchestrating the job’s work on allocated *containers* i.e., a slice of machine resources (some amount of CPU, RAM, disk, etc.). A client submits an AM package—that includes a
shell command and any files (i.e., binary executable configurations) needed to execute the command—to the RM, which then selects a single NM to host the AM. The chosen NM creates a shell environment that includes the file resources, and then executes the given shell command. The NM monitors the containers for resource usage and exit status, which the NM includes in its periodic reports to the RM. At runtime, the AM uses an RPC interface to request containers from the RM, and to ask the NMs that host its containers to launch a desired program. Returning to Figure 2.1, we see the AM instance running with allocated containers executing a job-specific task.

2.2 Big-data Analytic Applications

2.2.1 Distributed Machine Learning Algorithms

Machine learning has become the core of the big-data applications today. The huge success of Internet services such as data-mining, web-search and advertisement drives the rapid development of machine learning algorithms. In general, machine learning involves the training of a model based on a dataset. After training, the model can then be used to evaluate new datasets. The training is an iterative process, and the model is updated every iteration but the input data is shared between iterations. The machine learning applications today often involve huge amount of input data, so the training needs to be performed distributively over multiple nodes. Each node processes its local portion of data every iteration and communicates with other nodes to update the shared model. Therefore, distributed machine learning applications can be expressed in the form of MapReduce and will take advantage of the in-memory caching capability provided by Spark.

In this thesis, two of the most widely used distributed machine learning algorithms will be discussed: logistic regression and artificial neural network. Handwritten digit recognition, which involves classifying the input data as one of the ten digits from 0 to 9, has been chosen as an application case study.

Logistic regression [Fre09] is a combination of a linear model and a logistic function, which
regulates the output between 0 and 1. Therefore, it can be used in binomial classification where it provides a yes-or-no answer. The training is a gradient decent process and a very simple Spark program that performs logistic regression is shown below:

```scala
var points = // training input from HDFS
var weights = // random initial vector
for (i < 1 to ITERATIONS) {
  var gradient = points.map{
    p => p.x * (1/(1+exp(-p.y* (weights dot p.x))) - 1)*p.y
  }.reduce((a,b) => a+b)
  weights -= gradient
}
```

Figure 2.2: A two-layer artificial neural network topology used in data classification

Artificial neural networks model [HDB96] is derived from the perception system in the human brain. The model includes layers of neuron with weighted synaptic connections. Each neuron in a layer computes the weighted sum from all the neurons in the previous layer, and produces an output based on the sum and a sigmoid function. Figure 2.2 illustrates a basic two-layer neural network model used in classification, where the final output are the labels. The training of artificial neural networks is also a gradient decent process for the synaptic weights, which can be expressed in the form of MapReduce similar to logistic regression.
Figure 2.3: Circuit architecture. A) The model circuit consists of three sheets of grid cells, reciprocally connected with three VCO rings that receive driving inputs that encode movement velocity signals. The terms AMPA, NMDA, GABA indicates the types of synapses through which neurons are connected. B) Each VCO ring is composed of two circular layers of neurons, one excitatory and the other inhibitory, interconnected with one another by rotationally asymmetric weights so that an activity bump circulates around the ring in the counterclockwise direction. C) Spike rasters show propagation of the activity bump through the VCO layers (top two graphs) and the grid cell sheets (bottom graph) during a 3-second simulation of a rat running at constant speed along a linear path.

2.2.2 Spiking Neural Networks for Neuron Simulation

Large-scale SNNs are essential for a better understanding of the mystery behind the brain, and are increasingly being used to solve information processing tasks in an expanding range of engineering and robotic applications. In this thesis we discuss the general modeling method of neural microcircuits using SNN simulation with the example of circuits that conduct oscillatory path integration. Most animals (including humans) can keep track of their own location as they navigate through a familiar environment, even in darkness or when blindfolded. This ability depends upon path integration, the process of computing position by integrating movement velocity over time [OD71, ON78, HFM05, SBK08]. It has been hypothesized that these neurons derive their spatial tuning properties from neural oscillators with frequencies that are modulated by the speed and direction of an animal’s
movements through space [BBO07, BWZ07, GRZ07, GZF07].

An example architecture of the oscillatory microcircuit is illustrated in Fig. 2.3, which implements a spiking network for path integration by neural oscillators, as proposed by prior theoretical models. The circuit module is made up of two different circuits called VCO rings and grid sheets. Each circuit is composed from populations of single-compartment IAF neurons.

2.2.3 Convolutional Neural Networks for Image Recognition

Convolutional neural network (CNN) is a type of feed-forward artificial neural network that is inspired by the visual cortex, where the receptive neurons are aligned to respond to overlapping inputs of the visual field. In CNN, the neuron weights are shared for different regions of the visual input. Starting from LeNet [LBD89], variants of CNN have been delivering state-of-the-art accuracy in image recognition, and is prevalent in many computer vision contests including the notably ImageNet classification challenge [RDS15].

The typical CNN architecture is stacked convolution layers each optionally followed by rectified linear units (ReLU) layers and max-pooling layers, and one or more fully connected layers. For larger dataset like ImageNet, recent trend has been increasing the number of layers and the size of each layer, resulting in huge networks that have millions to billions of parameters. For the rest of this section we explain the basic notations of the CNN feed-forward networks.

Feed-forward: The feed-forward step of CNN is used to classify an input image. It is composed of multiple feature extraction layers, followed by classification layers. Each layer receives several feature maps from the previous layer and outputs a new set of feature maps. Figure 2.4 illustrates a general model of the feed-forward computation. The convolution layer, activation layer, and pooling layer are the main components in extraction layers; the fully connected layers are used in classification layers.

CONV: CONV layers are the main components of a CNN model. CONV computation is to extract feature information by adopting a filter on feature maps from previous layers. It
receives $N$ feature maps as input and outputs $M$ feature maps. During the computation of $N$ kernels, each sized in $K \times K$, the kernels slide across corresponding input feature maps with element-wise multiplication-accumulation to produce one output feature map. Assuming $b_{r,c}^m$ represents a pixel of the $m^{th}$ output feature map; $a_{r,y}^n$ represents a pixel of the $n^{th}$ input feature map; $\omega_{i,j}^{m,n}$ represents the weight in the convolution kernel between output $m$ and input $n$, and $S$ denotes the kernels’ sliding stride—then the computation of the CONV layer can be expressed as:

$$b_{r,c}^m = \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \omega_{i,j}^{m,n} \cdot a_{S \cdot r+S+i,S \cdot c+j}^n$$

(2.1)

**LRN:** LRN layers are sometimes applied after CONV layers. The LRN layer normalizes each pixel with pixels from $n$ neighboring feature maps at the same position. Assuming $b_{r,c}^m$ represents a pixel in the $m^{th}$ output feature map, and $a_{r,c}^m$ is a pixel in the $m^{th}$ input feature map, then the computation of the LRN layer can be written as:

$$b_{r,c}^m = a_{r,c}^m / (k + \alpha \cdot \sum_{j=\max(0,m-\frac{p}{2})}^{\min(N-1,m+p-1)} (a_{r,c+j}^i)^2)^\beta$$

(2.2)

where $k$, $\alpha$, $\beta$ are parameters calculated based on the training and validation dataset.

**POOL:** POOL layers are used to achieve spatial invariance by sub-sampling neighboring pixels, normally finding the max value in a neighborhood in each input feature map.

**NL:** NL layers are used to adopt a non-linear function on each pixel of feature maps from previous layers to mimic the biological neuron’s activation.
**FC:** FC layers are used to make final predictions. The FC layer takes “features” in a form of vectors, from feature extraction layers and outputs a new feature vector. Its computation pattern is exactly a dense matrix-vector multiplication or inner-product. A few cascaded inner-products finally output the classification result of CNN. Assuming $a^n$ and $b^m$ represent elements in the $n^{th}$ input feature and $m^{th}$ output feature respectively and $\omega^{m,n}$ represents the weights, then the computation of the FC layer can be written as:

\[
b^m = \sum_{n=0}^{N-1} \omega^{m,n} \cdot a^n
\]  

(2.3)

The most compute intensive part of both CNN training and testing are the convolution layers, which can take more than 90% of the total computation [CX14]. Due to its regular computation patterns, GPUs acceleration have been dominant in most CNN implementations, with many optimized framework such as cuDNN [CWV14] and Caffe [JSD14]. Due to its popularity, many customize hardware architectures have been proposed as well for better energy-efficiency [ZLS15]
In this chapter, we introduce the scale-up design methodology focusing on improving the programmability and performance of accelerators for a single application.

3.1 Overview

With the recent advance of FPGA HLS [CLN11], the programming complexity of an accelerator is greatly reduced. However, programming with HLS still requires detailed hardware-specific knowledge, so it is unrealistic for domain experts in big-data analytics to leverage the performance and energy-efficiency provided by FPGA-based accelerators. For FPGA experts that can program with HLS or RTL, the fast-evolving nature of big-data analytics make it unrealistic to cater to all the algorithm-level modifications.

Many big-data analytics applications have well-formed structures and fixed computation patterns. Therefore, domain-specific languages can be used to generate efficient accelerator design without hardware knowledge. Furthermore, raising the program abstraction level also creates opportunities to explore the design space for performance and energy-efficiency optimization. In this chapter, methodologies of synthesizing efficient FPGA engines from domain-specific languages are presented for two of the widely popular brain-inspired models: spiking neural networks (SNNs) and convolutional neural networks (CNNs). Both models have fixed computation patterns and can reach very large scale. The application of SNN described in this chapter is neural microcircuits simulation, and the application of CNN is image classification.

For the SNN engines, a language based on XML is created to describe a network. For the
CNN engines, the existing model description from Caffe [JSD14] is adopted. For both models, domain-specific language descriptions are used to synthesize basic compute units. The compute units are then reconfigured through loop transformation, pipelining and unrolling to optimize system throughput. Although the processing flow of the SNN engines and CNN engines are different (SNN is a synchronized processing and CNN is a streamed processing), the optimization goal for both engines is matching the throughput of all compute units.

In this thesis, a single-node solution for the SNN simulation engine is presented. It will be extended to multi-node using the same design for the CNN compute engine. For the CNN compute engine, an automated flow that synthesizes the Caffe-based CNN model description into multi-node FPGA systems is proposed.

3.2 FPGA Simulation Engine for Spiking Neurons

In this section, a platform-based methodology that uses Xilinx Vivado high-level synthesis tools [viv] for constructing FPGA simulations of neural microcircuits is presented.

Based on the description in Section 2.2.2, in SNN simulations, each neuron cell is modeled as one or more compartments; neuron cells sharing the same properties form a population; and populations of neurons form circuits. Neuron compartments are modeled as electrical circuits that represent membrane voltage, current and conductance. A neural microcircuit is populations of neurons connected together through spiking events that increase or decrease membrane voltage through synaptic conductance.

Exploiting such hierarchical structure, an XML-based description for neural microcircuits is designed and used to generate C codes for high-level synthesis tools can be generated using design templates. In the proposed platform, neuron populations are mapped to computing engines, in which the neuron cells are simulated with integration pipelines. The proposed platform-based methodology has the following advantages:

- The proposed approach enables rapid construction of FPGA simulation engines throughout the domain of neuron simulation. Taking advantages of the hierarchical structure
shared by most spiking neuron models, our platform can generate efficient hardware
implementations with design templates and high-level synthesis tools.

- Based on high-level description of neural microcircuits, the platform enables the domain
  experts in neural modeling to synthesize high-performance and energy-efficient neuron
  simulation engines without extensive knowledge of the FPGA design details. When
  creating or modifying a neuron simulation, the model configurations made by domain
  experts can be automatically mapped to the corresponding FPGA implementation.

- With the platform, the task of exploring design trade-offs and performance optimization
  is also simplified. Optimizations can be performed at each level of the simulation
  without violating the semantics of the model.

3.2.1 Neural Microcircuits Modeling

A neural microcircuit can be expressed with a hierarchical structure: compartments, cells,
populations and circuits. The higher-level objects are composed from the lower-level com-
ponents. In the next part of this chapter, each level of the neuron model is discussed in
details.

3.2.1.1 Compartment

A neuron compartment is the basic unit of neuron simulation, which models a section of a
neuron cell as an electrical circuit illustrated in Fig. 3.1. In the circuit, a series of synaptic
channels modeled by a reversal potential $E$ and conductances $g$ are connected in parallel
with the membrane voltage $V_m$ and the membrane capacitance $C_m$. The electrical circuit is
simulated by numerical integration with the forward Euler method.

The synaptic channels receive the spiking stimulus from pre-synaptic neurons that changes
their conductances. As a result, the membrane voltage will increase or decrease based on the
activation of synaptic channel and the driving force. If a synapse decreases the membrane
voltage, it is called an inhibitory neuron, while if the neuron increases the membrane voltage,
it is called an excitatory neuron.

The synaptic conductances obey first-order kinetics, and are updated at each time step by numerical integration as follows:

$$g(t+1) = g(t) - \frac{1}{\tau_a}g(t) + A\delta(t) \quad (3.1)$$

where $g(t)$ is the channel conductance at time step $t$, and the decay of conductance is modeled as time constant $\tau_a$. The strength for each type of synapse is scaled by $A$, and the spiking event is modeled with a delta function where $\delta(t) \neq 0$ when a pre-synaptic spike is fired at time step $t$.

Eq. 3.1 describes the basic model for AMPA and GABA synapses. The kinetics model for NMDA synapses is slightly different, but it is mapped to our platform in the same way. Therefore, this level of details are not elaborated in this thesis.

3.2.1.2 Cell

A neuron cell contains one or more compartments. In the case of the oscillatory neural microcircuit, all cells are modeled with single-compartment.

In the IAF neuron model, a neuron spikes when $V_m$ reaches a certain threshold $V_{\text{thresh}}$; after spiking, $V_m$ is held at a fixed reset potential, $V_{\text{reset}}$ for a certain period, which is called the post-spike refractory period. The refractory period is updated each time step by:

$$R(t+1) = \begin{cases} T_{\text{refract}}, & \text{if } V_m(t) > V_{\text{thresh}} \\ R(t) - 1, & \text{otherwise} \end{cases} \quad (3.2)$$
where $V_{\text{thresh}}$ is the spike threshold and $T_{\text{refract}}$ is the number of the time steps in the refractory period.

After the refractory period, $V_m$ is updated by numeric integration using the Forward Euler method:

$$V_m(t+1) = V_m(t) + \lambda (I_{\text{in}}(t) - I_{\text{out}}(t)) \quad (3.3)$$

where $t$ is the simulation time step, $I_{\text{in}}$ and $I_{\text{out}}$ are the total inward (excitatory) and outward (inhibitory) membrane currents. $\lambda$ here is the constant value that models the membrane capacitance $C_m$ and time step $dt$.

The membrane current $I_{\text{in}}$ and $I_{\text{out}}$ is the sum of currents $i$ of each channel, which is computed as the product of a channel conductance $g$, and driving force $D$.

$$i(t) = D(t) * g(t) \quad (3.4)$$

The driving forces are computed based on the membrane voltages and the reversal potentials of each channel.

### 3.2.1.3 Population

A neuron population is a group of neurons that share the same cell model and regular structured connections. In neuron simulations, having such an intermediate layer is important for efficiency.

As established in the previous section, for each synapse there will be a corresponding channel in the post-synaptic neuron, with a reversal potential and conductance. It is very costly to keep track of all these state variables, especially when neurons often have a large fan-in. For example, in the oscillatory path integration circuit a grid cell has about 600 synapses. However, when a group of neurons share the same type of synaptic channel, it is possible to combine these separate state variables into one.

When neurons in a population are connected to all neurons in another population, the synaptic channels for those connections in a post-synaptic neuron can be represented as a
population conductance $G(t)$, which is the weighted sum over corresponding conductances.

$$G_i(t) = \sum_j \omega_{i,j} \cdot g_{i,j}(t) \quad (3.5)$$

where $i$ indexes the neuron in the post-synaptic population, and $j$ indexes neurons in the pre-synaptic population. $\omega_{i,j}$ is the synaptic weight coefficient representing the strength of the connection between neuron $i$ and $j$.

With the aggregation of synaptic channels, the amount of computation to simulate a population can also be reduced because only spiking neurons will change the activities of connected neurons. This observation can be made from combining Eq. 3.1 with Eq. 3.5.

$$G_i(t+1) = \sum_j \omega_{i,j} \cdot (g_{i,j}(t) - \frac{1}{\tau_a} g_{i,j}(t) + A \delta_j(t))$$

$$= G_i(t) - \frac{1}{\tau_a} g_i(t) + \sum_k A \cdot \omega_{i,k} \quad (3.6)$$

The first two terms of Eq. 3.6 represent a decay of the conductance, while the third term represents a spiking stimulus, where $\sum_k A \cdot \omega_{i,k}$ is the summation over all spiked neurons in the pre-synaptic population at time step $t$.

The weights $(\omega_{i,j})$ can also be aggregated as weight vectors of predefined values. In the case of the oscillatory neural microcircuits, the weight vector is defined by a Gaussian distribution [SW05].

$$\omega_{i,j} = W(i - j) = C \cdot \exp\left(\frac{\cos(\theta * (i - j) - \theta_0)}{\sigma^2}\right) \quad (3.7)$$

where $C$ is a normalizing factor, and $\theta *(i - j)$ represents the phase difference between neuron $i$ and $j$ in respect to the VCO ring structure.

### 3.2.1.4 Circuit

A neural circuit is a collection of one or more neuron populations that can produce particular spiking patterns. Creating neuron populations and connecting them in a certain way to create circuits that generate required spiking patterns is the basic task of neuron modelers. Therefore, for a hardware accelerator platform performing neuron simulation, it is important to provide flexibility to both the neuron diversity and connectivity.
An example model of the oscillatory neural microcircuits is illustrated in Fig. 2.3. The whole module is connected by two types of circuits: VCO rings and grid sheets. Each VCO ring includes two populations of neurons: inhibitory neurons (inh) and excitatory neurons (exc). These two populations of neurons are interconnected with each other, and the inhibitory layer is reciprocally connected to itself. The excitatory population also receives external excitatory spike input from velocity cells (ext). The connections from inhibitory to excitatory cells are asymmetric, so that the neurons in the VCO ring fire sequentially as a ‘bump’ of spiking activity circulates around the ring at a certain frequency [BWZ07, WSB11, SW05]. Each grid sheet includes one population of grid cells (grid) that connects to three VCO rings by receiving excitatory and inhibitory spikes from the neurons at different angular positions in those rings.

### 3.2.2 Platform-Based FPGA Implementation

![Diagram of neural microcircuit implementation](image)

Figure 3.2: The mapping of the neural microcircuit to our platform. The left side of the figure illustrates the hierarchies of the neural microcircuit and the examples of each level in the oscillatory model. The right side illustrates the corresponding implementation in our platform-based approach.

Fig. 3.2 illustrates the mapping between a neural microcircuit and the FPGA implementation with our platform. The mapping is described in a high-level XML file, in which
domain experts specify the types of synaptic channels in each cell, the spiking neuron model, the populations, and the synaptic connections between the populations. Then the XML description is used to generate C code specified to FPGA implementation, and the C code will be synthesized into RTL using high-level synthesis tools. Performance tuning parameters such as loop unrolling factors can be expressed on the high-level XML description. For FPGA design experts, design optimization can also be explored on the C level.

3.2.2.1 Mapping Neuron Compartments and Cells

In our platform, neuron compartments are modeled with a spiking model and a synapse model. The spiking model describes the integration of membrane voltages ($V_m$) of each compartment. A spiking mechanism such as the IAF model is also defined in the cell model in our platform. The synapse model describes the integration of synaptic conductances $G$. Both multiple-compartment cells and single-compartment cells can be modeled using the same electrical circuit model in Fig. 3.1.

The following XML code is an example of describing a single-compartment neuron cell with the basic spiking model. Based on the definition of the cell type, which in this piece of code is IAF, our platform generates integration operations based on the biological model described by Eq. 3.2 and Eq. 3.3 from pre-defined templates. The property tags vreset, vthresh and trefract configure $V_{reset}$, $V_{thresh}$ and $T_{refract}$ respectively. This description in the example assumes that “iafreset”, “iafthresh” and “rstep_exc” will be defined by the user after the C code for high-level synthesis is generated, which is the same goes for the examples afterwards.

```xml
<cell type='IAF'>
  <vreset>iafreset</vreset>
  <vthresh>iafthresh</vthresh>
  <trefract>rstep_exc</trefrac>
</cell>
```

The synaptic channels in each compartment are modeled by describing a synapse on the
population level. The code below shows the description of a GABA synapse called *inhexc*. By defining a synapse, our platform creates a synaptic conductance in the post-synaptic population, and a connection between the pre-synaptic and post-synaptic population. Our platform supports the basic types of synapses (AMPA, NMDA, GABA) by having templates for the specific types of integration operations used to model these synapses.

```xml
<synapse name='inhexc'>
  <pre>inh</pre>
  <type E='egaba' A='4096' tau='tau_gaba'>GABA</type>
  <weights>FUNC</weights>
</synapse>
```

In our platform, we define the synapses in the post-synaptic population, while the tag `<pre>` in the synapse description indicates the pre-synaptic population. Tag `<type>` indicates the specific update routine of this synaptic channel, and `<weights>` specifies the type of weight models that represent the strength of the synapse. This will be elaborated in the next section.

### 3.2.2.2 Mapping Neuron Populations

As described in Section 3.2.1.3, the definition of neuron populations enables a variety of simplifications for simulation. Our platform exploits these simplifications by mapping a neuron population to a computation engine that contains two update pipelines: a conductance update pipeline and a membrane voltage update pipeline. In the pipelines, state variables of each neuron are integrated iteratively. An outline design of the computing engine structure is illustrated in Fig. 3.3.

The conductance update pipeline integrates synaptic conductances based on the synapse model described in Section 3.2.2.1. The pipeline can be divided into two stages—a decay stage that is irrelevant to the spiking input, and a spiking stage that is only activated when there is a spike input. Such a division follows the description in Eq. 3.6, where the spiking stage calculates the weighted sum over stimulus generated from the spiking neurons.
The weights $\omega_{i,j}$ in Eq. 3.6 are modeled as weight vectors between two populations. Our platform supports four kinds of weight vectors:

1. Constant weights, where all $\omega_{i,j}$ are constant for post-synaptic neuron $i$

2. Direct functional weight mapping, where the weight values depend on the simple distance between the two neurons $i$ and $j$

$$\omega_{i,j} = f_\omega(i - j) \quad (3.8)$$

3. Indirect functional weight mapping, where the weight values depend on the distance of neuron $i$ and $j$ that is defined in a distance vector

$$\omega_{i,j} = f_\omega(f_{dist}(i,j)) \quad (3.9)$$

4. Direct mapping, where a separate weight is defined for each pair of neurons.

The following code snippet shows an example of the description of a population that is named as exc and has a size of 120 neurons.

```xml
<population name='exc' size='120'>
  <cell type='IAF'>
    ...
  </cell>
  ...
</synapse name='inhexc'>
In the computing engine, neuron local states are stored in memory blocks (BRAM of FPGA), and by default the two update pipelines integrate each neuron sequentially. For all the update pipelines in the template, the *Initiation Interval (II)* is optimized to 1, so that optimal throughput is achieved. To further improve performance, our platform also supports unrolling of the pipelines, which means neurons in a population can be updated in parallel by a certain factor. The developer can specify such factor in the `<population>` tag, and our platform will partition the corresponding memory blocks automatically based on the specification. The loop pipelining and unrolling is implemented by using pragmas in the Xilinx Vivado HLS tool.

### 3.2.2.3 Mapping Neural Microcircuits

Previously we introduced the computing engine and its basic structure. Here, the focus is on the mapping between system-level composition of neural microcircuits and the interconnection of computing engines in our platform.

```xml
<circuit name='vcoring' num_popl='2'>
  <population name='exc' size='120'>
    ...
  </population>
  <population name='inh' size='120'>
    ...
  </population>
</circuit>
```

The code snippet above shows an example of the circuit called `vcoring`, containing two populations: `exc` and `inh`. With the circuit structure described in the XML file, our platform...
generates C code for each computing engine. The connections between pre-synaptic populations and post-synaptic populations are defined by each synapse description, with which our platform generates the system-level interconnections between the computing engines. The platform also generates a top module that includes the necessary interfaces.

On the system level, our platform simulates neuron populations synchronously. That is, all computing engines synchronize at each time step. In each time step, all computing engines perform the conductance update first, and then the voltage update pipeline can integrate the membrane voltages and generate spiking events. After all spiking events of each population have been generated, the simulation proceeds to the next time step so that the spiking events can be used in the conductance update pipelines. This approach is based on the Forward Euler integration algorithm used by our simulation. In this thesis, we do not explore other integration methods, since the Forward Euler is one of the simplest and most popular algorithm used by neural modelers.

3.2.2.4 Case Study: Oscillatory Neural Microcircuits

Here we discuss the implementation of the oscillatory neural microcircuit based on our platform design as case study. We start from the description of a single VCO ring, which is one of the components in the oscillatory neural microcircuit. Following the description in Section 3.2.1.4, the inh and exc populations are connected by NMDA and GABA synapses, and the weights are modeled as Gaussian distribution in Eq. 3.7. Therefore, we use the direct functional mapping to describe the weight vectors in our platform. The exc population also receives external excitatory signals as a Poisson spike train through AMPA synapses, which is modeled with a random number generator in the conductance update pipeline, and the weight vector type is constant. The XML description of a VCO ring circuit based on our platform is already introduced as examples previously.

The full grid module of the oscillatory neural microcircuit is composed of two types of circuits: VCO rings and grid sheets. Besides the exc and inh populations that form the VCO ring, there is another type of population called grid that forms the grid sheets. The
circuit contains nine populations in total. To describe the circuit in our platform, the user just need to define the nine populations with the basic cell type and the synapses describing the connections. The IAF model used in all populations and the three types of synapses (AMPA, GABA and NMDA) are supported in our platform by templates. We use indirect functional mapping to model the weight vectors for synapses between neurons in the VCO rings and neurons in the grid sheets.

As described in Section 3.2.2.2, performance of the simulation engine can be improved by defining unrolling factors for each population. For the VCO ring, the choice is straightforward because the two populations have the same size. Therefore, the performance will increase by unrolling the two populations with the same factor. We presents the experimental results for one VCO ring in Section 3.2.3.1. For the full grid module, on the other hand, design choices become more complicated. Based on the performance bottleneck of the implementation, decisions of unrolling which population have different impacts on the performance. We show the discussion on the design space exploration and optimization in Section 3.2.3.2.

We also briefly discuss the scalability of our platform with the calculation of the bandwidth requirement of the oscillatory neural model in this section. As illustrated in Figure 2.3, each VCO ring is connected to two adjacent grid sheets with excitatory links, and to the other grid sheet with an inhibitory synaptic link. The oscillatory neural microcircuit is scaled up by adding the VCO ring and grid sheet in pairs. Because the role of the grid sheets is to detect synchrony between two VCO rings, the number of total connections grows linearly to the number of rings and grid sheets. Assuming 32 bits are used to index the location of each fired neuron, which is more than enough for most neural microcircuits, the bandwidth requirement for each link between computing engines is:

\[ B = 4N \ast \alpha/t_s \]  

(3.10)

where \( B \) is the bandwidth in Byte/second, \( N \) is the number of neurons mapped to the computing engine, \( \alpha \) stands for the firing rate, and \( t_s \) is the simulation time step. Based on the equation, the roughly estimated bandwidth requirement for a single link is 2MB/s with firing rate of 10%, which is a loose upper bound for most neural circuits. The aggregated
bandwidth for the entire microcircuit is around 30MB/s.

In our platform-based approach, the performance of each computing engine depends on the number of neurons in the simulated population and the firing rate of neurons in its connected computing engines. The number of connections between two populations does not grow as the model scales up. So the overall system performance is roughly proportional to the number of neurons in the system. Therefore, the performance is scalable by adding computing engines to the system such that the number of neurons mapped to each computing engine remains the same. Therefore, when the computing engines need to be mapped to multiple chips, the inter-chip connection bandwidth should not be a performance bottleneck.

3.2.3 Experimental Results

Here we demonstrate the performance of our FPGA simulation engine for neural circuits that perform oscillatory path integration. The results show that our platform-based approach is capable of simulating a range of different neuron models with good performance. The FPGA implementation based on our platform achieves up to 39x and 264x speedup compared to the Intel CPU and embedded ARM A9 processor respectively. We also analyze the energy-efficiency of our platform in comparison with CPU and ARM, as well as a selection of existing implementations. Compared to pure software implementation on an embedded ARM core, our platform achieves 232x energy reduction.

The CPU benchmark is the original sequential code used in oscillatory path integrator simulation, and was measured on an Intel Xeon L5408 CPU. The embedded CPU experiments were used on an ARM A9 core in a Xilinx Zynq-7045 SoC chip. The FPGA platforms for our hardware simulation evaluation is the Zynq-7045 (xc7z045ffg900-2) FPGA running at 100MHz clock.

3.2.3.1 Case Study: One VCO ring

We first present the performance evaluation of our computing engine design with the simulation of a single VCO ring. In the experiments, each of the two neuron populations (inh and
"exc" consists of 120 IAF neurons. In total, one VCO ring has 240 IAF neurons and 43,200 synapses.

In Table 3.1, the performance of simulation engines generated by our platform with different unrolling factors is presented. The symbol "\([i:j]\)" indicates the unrolling factors for the computing engines for \(inh\) and \(exc\) respectively. For instance, "FPGA [4:4]" represents the implementation in which the \(inh\) and \(exc\) computing engines are unrolled for four times. As described in Section 3.2.2.2, the unrolling factor can be defined in each population. Since the two populations of the VCO ring has the same amount of neurons, we always unroll the two population with the same factor. The CPU and ARM performance is measured as single-thread software implementations.

Table 3.1: Performance of one-second simulation of a single VCO Ring

<table>
<thead>
<tr>
<th>Platform</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>0.65s</td>
<td>1</td>
</tr>
<tr>
<td>ARM</td>
<td>6.65s</td>
<td>0.1x</td>
</tr>
<tr>
<td>FPGA [1:1]</td>
<td>0.137s</td>
<td>4.7x</td>
</tr>
<tr>
<td>FPGA [2:2]</td>
<td>0.084s</td>
<td>7.7x</td>
</tr>
<tr>
<td>FPGA [4:4]</td>
<td>0.057s</td>
<td>11x</td>
</tr>
</tbody>
</table>

Table 3.2 shows the trade-off between the resource usage and performance for designs with different unrolling factors.

Table 3.2: Resource usage of FPGA Implementations of one VCO Ring

<table>
<thead>
<tr>
<th></th>
<th>BRAM</th>
<th>FF</th>
<th>LUT</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA [1:1]</td>
<td>18</td>
<td>4.9k</td>
<td>5.3k</td>
<td>4.7x</td>
</tr>
<tr>
<td>FPGA [2:2]</td>
<td>40</td>
<td>11.7k</td>
<td>12.5k</td>
<td>7.7x</td>
</tr>
<tr>
<td>FPGA [4:4]</td>
<td>78</td>
<td>16.2k</td>
<td>18.5k</td>
<td>11x</td>
</tr>
</tbody>
</table>
3.2.3.2 Case Study: Oscillatory Grid Module

The experiments of the grid module simulation are conducted with the configuration that each population in the VCO ring contains 108 neurons, and each grid sheet contains 16 neurons. In total, the grid module contains 756 neurons interconnected by 174,960 synapses.

Because of the heterogeneity of the three populations, design choices can be made on unrolling different computing engines. Intuitively, computing engines that simulate the grid population is not the performance bottleneck, since it has far less neurons than the populations in the VCO rings. We discuss the decisions on optimizing performance in Table 3.3. The symbol “[i:j:k]” for FPGA implementation indicates the unrolling factors for the computing engines for \( inh \), \( exc \) and \( grid \) respectively. For instance, “FPGA [4:4:2]” represents the design in which the computing engines for \( inh \) and \( exc \) are unrolled with a factor of 4, while the computing engines for \( grid \) are unrolled with a factor of 2.

Table 3.3: Performance of one-second-simulation of the Grid Module

<table>
<thead>
<tr>
<th>Platform</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>3.69s</td>
<td>1</td>
</tr>
<tr>
<td>ARM</td>
<td>24.83s</td>
<td>0.15x</td>
</tr>
<tr>
<td>FPGA [1:1:1]</td>
<td>0.142s</td>
<td>26x</td>
</tr>
<tr>
<td>FPGA [2:2:1]</td>
<td>0.105s</td>
<td>35x</td>
</tr>
<tr>
<td>FPGA [4:4:1]</td>
<td>0.101s</td>
<td>36.5x</td>
</tr>
<tr>
<td>FPGA [4:4:2]</td>
<td>0.094s</td>
<td>39x</td>
</tr>
</tbody>
</table>

The results in Table 3.3 shows that the performance increases with larger unrolling factors of the populations in the VCO rings. However, the performance increase becomes small when reaching “FPGA [4:4:1]”. That is because the after the simulation of the VCO rings become faster, the grids becomes the performance bottleneck. Therefore, in “FPGA [4:4:2]” the performance increases when the \( grid \) is unrolled for two times.

In Table 3.4 we present the resource usage for different implementations. The resource almost doubles when the VCO rings are being unrolled, since the computing engines for
Table 3.4: Resource usage of FPGA Implementations of the Grid Module

<table>
<thead>
<tr>
<th></th>
<th>BRAM</th>
<th>FF</th>
<th>LUT</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA [1:1:1]</td>
<td>57</td>
<td>15.4k</td>
<td>17.1k</td>
<td>26x</td>
</tr>
<tr>
<td>FPGA [2:2:1]</td>
<td>86</td>
<td>37.7k</td>
<td>35.8k</td>
<td>35x</td>
</tr>
<tr>
<td>FPGA [4:4:1]</td>
<td>145</td>
<td>74.9k</td>
<td>70.2k</td>
<td>36.5x</td>
</tr>
<tr>
<td>FPGA [4:4:2]</td>
<td>148</td>
<td>78.0k</td>
<td>73.0k</td>
<td>39x</td>
</tr>
</tbody>
</table>

\(inh\) and \(exc\) are considerably larger than \(grid\). Design choices can be made based on the trade-offs between performance and resource.

3.2.3.3 Energy Consumption Estimation

To compare the energy performance of our FPGA simulation engine with other established implementations, we first define the metric that will be used to evaluate energy consumption. Since the existing work implements different neuron models, it is unfair to compare the overall power of the simulation. One plausible metric can be the unit energy consumption for a single spiking. Such a metric can be found in [MAA11, SGR12].

We first look at the energy consumption of the high-performance clusters. IBM did not provide the energy statistic for their BlueGene simulator [AES09]. Nevertheless, [SGR12] provides an estimate that it “consumes 655kW to simulate \(1.6 \times 10^9\) neurons at the speed of 643 second per \(Hz\).” So the energy per spiking-event is \(655 \times 10^3 \times 643/(1.6 \times 10^9) = 263.6mJ/s.e.\) The SpiNNaker system, which uses a multi-core ARM system with customized interconnection, consumes \(43nJ/s.e.\) [SGR12]. On the other end, IBM’s crossbar-based ASIC implementation consumes \(45pJ/s.e.\) [MAA11].

Our energy per spiking-event can be derived using the overall power consumed to simulate one second of neuron behavior and the average spike-rate per second. The implementation we used in this experiment is the grid module design “FPGA \[4:4:2\]”. The spike-rate is measured from simulation, which is about 120k spikes for the entire grid module in a one-second simulation. The energy consumption of CPU is estimated by operating power (40W
for Xeon L5408, 10W for one core) times the simulation time. The power of ARM and FPGA implementation is recorded using the Texas Instrument Fusion Technology, which monitors real-time voltage and current data using the power buses on our Xilinx ZC706 board. Table 3.5 shows the energy per spiking-event statistics for the implementation of the Oscillatory Grid Module.

Table 3.5: Energy Estimation of Simulating Grid Module on Different Platforms

<table>
<thead>
<tr>
<th>Platform</th>
<th>Total Energy</th>
<th>Energy per Spiking-event</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>36.9 J</td>
<td>0.308 mJ/s.e.</td>
</tr>
<tr>
<td>ARM</td>
<td>8.03 J</td>
<td>66.9 µJ/s.e.</td>
</tr>
<tr>
<td>FPGA</td>
<td>34.5 µJ</td>
<td>0.288 µJ/s.e.</td>
</tr>
</tbody>
</table>

Since the ARM core in the Zynq chip does not turn off when idle, the energy measurement for FPGA includes the energy cost of the ARM core as well. However, with the considerable performance improvement of hardware acceleration, our platform-based implementation still achieves 232x less energy than the embedded solution based on ARM.

3.3 Multi-FPGA Acceleration Platform for Convolutional Neural Networks

Along with the evolution of CNN models, a general trend is to scale up both the network size and computation complexity. To satisfy the growing demand on computation capability, researchers have used or created various high-performance hardware platforms, including GPGPU or customized accelerators such as FPGA and ASIC to improve performance and efficiency [JSD14, CDS14, CLL14, SJC09, ZLS15].

FPGA-based accelerators have been gaining popularity in accelerating large-scale CNN models because they can achieve lower latency and consume much less power compared to GPGPUs; they are also more flexible compared to ASICs. They are especially favored in datacenter-scale deployments [OLQ14]. Previous work mostly focused on single-board im-
plementation [ZLS15, FPH09, CSJ10b, QWY16, SCD16]. However, the single-board design has several inefficiencies. First, FPGAs typically have fewer on-chip floating-point units and memory buffers, which limits their overall computing power. Second, the compute kernels in a CNN model can have different requirements for compute resource, memory bandwidth and on-chip buffers. As a result, it is extremely difficult to balance the resource allocation on a single FPGA for different layers in the CNN.

Table 3.6: Computation and memory complexity of different CNN layers in AlexNet [KSH12]

<table>
<thead>
<tr>
<th></th>
<th>CONV</th>
<th>LRN</th>
<th>POOL</th>
<th>NL</th>
<th>FC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total OPs (10^9)</td>
<td>591</td>
<td>5.5</td>
<td>0.5</td>
<td>0.0</td>
<td>37.7</td>
</tr>
<tr>
<td>Percentage</td>
<td>93.2%</td>
<td>0.8%</td>
<td>0.1%</td>
<td>0.0%</td>
<td>6%</td>
</tr>
<tr>
<td>Weight Size (MB)</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>224</td>
</tr>
<tr>
<td>Percentage</td>
<td>4.31%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>95.7%</td>
</tr>
<tr>
<td>DSPs per OP</td>
<td>1-5</td>
<td>11</td>
<td>3</td>
<td>0</td>
<td>1-5</td>
</tr>
</tbody>
</table>

The second issue can be better explained in Table 3.6, which is a detailed analysis of the feed-forward stage of layers in AlexNet [KSH12], one of the most studied CNN models that won the 2012 ImageNet contest. From Table 3.6, the following conclusions can be drawn:

*Convolution (CONV) layers are compute-intensive; they consume 4.31% of the total weights but occupy more than 93.2% of the total arithmetic operations. Fully-connected (FC) layers are memory bandwidth intensive; they contain only 6% of all the arithmetic operations but require 95.7% of the total weights. Local response normalization (LRN) layers are resource-costly; although they contain a small amount of arithmetic operations, it requires a lot of FPGA’s DSP resources due to its power operations. Pooling (POOL) and Non-linear (NL) layers do not consume much computing power or memory bandwidth, and do not require a large amount of FPGA resources.*

Focusing on this issue, a deeply pipelined multi-FPGA architecture to accelerate the feed-forward stage of large-scale CNNs is presented. We also build a prototype of seven FPGA boards connected with high-speed serial links in a ring network to demonstrate the performance and energy efficiency of the proposed architecture. To partition an existing CNN model on the system efficiency, we propose a dynamic programming algorithm to explore the
optimal mapping of each CNN layer on different FPGAs. In summary, the proposed work in this chapter has the following contributions:

- We propose a quantitative model for mapping CNNs to the FPGA cluster. To our best knowledge, we are the first to optimally explore the design space of mapping CNNs on multiple FPGAs.
- We develop a prototype system to demonstrate that a deeply pipelined FPGA cluster can achieve comparable performance to implementations on a high-end GPU while consuming much less energy.

### 3.3.1 Deeply Pipelined FPGA Cluster and Prototype System Design

The computation of the feed-forward phase of a CNN model is a streaming flow from the first layer to the last based on the description in Section 2. As a result, in the proposed architecture of the deeply pipelined FPGA cluster, multiple FPGAs are connected in a ring network, which is illustrated in Figure 3.4(a). On each of the FPGAs, a computation engine is customized for a specific one or more CNN layers according to optimization methods in Section 3.3.2. High-speed serial links are used as an interconnection between two FPGA boards, and a FIFO-based protocol is implemented on the inter-FPGA links.

#### 3.3.1.1 Interconnection

Low latency and high bandwidth are the requirements for the inter-FPGA connection due to the large volume of input and output data for each CNN layer, especially when the cluster is processing images in batches. A bidirectional board-to-board connection using the FPGA high-speed serial transceiver and receiver is implemented using the Xilinx Aurora protocol [aur]. Each link has a bandwidth of 750MB/s, and the latency is typically around 100 FPGA cycles.

We also implemented a low-overhead flow control protocol to avoid overflow of the input buffer of each FPGA. In our protocol, the buffer state of FPGA II is back-propagated to
FPGA I using the same serial link to transfer data.

### 3.3.1.2 Computation Kernels

Figure 3.4(b) illustrates the mapping of CNN computation engines on each FPGA. We build three computation engine models for the five kernels. Based on the computation patterns of CNN kernels, we use multiple channels to provide concurrent data streams to an SIMD computation engine for parallel processing. The number of channels are parameterized and constrained by FPGA on-chip BRAM and DSP number. The overall design space exploration methodology is described in Section 3.3.2.

*CONV* and *FC* layers can both be accelerated on a generalized CONV engine. For CONV layers, we make a parallelism model of $T_n$ and $T_m$ channels for input and output feature maps, respectively. FC layers are data-intensive and are usually bounded by bandwidth for FPGA platforms. We solve this problem by batching the input feature maps for FC, which enables the sharing of the weight matrix.

*POOL* and *NL* layers can usually be merged with the computations of CONV layers
to mitigate the latency and resource cost. Therefore, the POOL and NL engines have $T_m$ parallel channels, which is the same as the output of CONV engines.

LRN layers are DSP resource-costly. An improper resource allocation for LRN layers can make it the bottleneck of the system and degrade overall performance significantly. This can be illustrated by the results in Section 3.3.3. In the LRN engine, We model a partitioning of $T_l$ parallel PEs for LRN layers.

To demonstrate the performance and energy-efficiency of the proposed architecture, we build a hardware prototype consisting of six Xilinx VC709 boards. The prototype system is designed for the detection task of the ImageNet large scale visual recognition challenge (ILSVRC) [RDS15], with the objective of getting a reasonable detection accuracy with significant lower energy than traditional high-performance systems. A snapshot of the system is shown in Figure 3.5b.

The algorithm used for image detection is based on RCNN [GDD14], which divides the detection task into two steps: region proposal and region classification. The general idea of the algorithm is first proposing as many candidate regions as possible, and use a CNN to extract feature vectors for each region. Finally, all the feature vectors are then classified and pooled for a final answer. In the prototype implementation, the region proposal algorithm is based on [USG13]. The CNN for feature extraction is the classic AlexNet [KSH12].

In the prototype cluster, there are seven FPGA boards in total. Among the seven, one board is a Xilinx ZC706 board, which has the Zynq SoC of ARM cores and FPGA fabrics. This Zynq board is used for region proposal. After extracting the regions from an image, the Zynq board sends all the image region patches to the subsequent boards through high-speed serial links. The six subsequent boards are Xilinx VC709 boards, each of which has large amount of reconfigurable resources. These VC709 boards are connected through high-speed serial links as well, and they are implemented as a processing pipeline of the CNN extraction layer by layer. The FPGA implementation of AlexNet is based on [ZLS15]. The layout of the prototype system is illustrated in Figure 3.5a. The computation engines on each boards are duplicated and optimized to maximize system throughput.
Table 3.7 shows the comparison of different GPU platforms and the prototype FPGA cluster. The GPU CNN performance is based on reports of existing implementation based on Caffe, and the detection performance is estimated based on a hand-tuned C++ implementation of the RCNN pipeline. Even though the prototype FPGA cluster does not the optimal performance and the initial version uses a slow clock frequency for stability, its energy efficiency for classification is only 2x or 3x behind the optimal GPU implementation, and the detection energy efficiency exceeds all the GPU platforms in the table.

Figure 3.5: The prototype system for the scale-up design
Table 3.7: CNN Performance and energy results of different platforms

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>K40</td>
<td>235</td>
<td>823.7</td>
<td>1.01</td>
<td>3.51</td>
<td>3.03</td>
</tr>
<tr>
<td>GTX 770</td>
<td>230</td>
<td>480.8</td>
<td>0.75</td>
<td>2.09</td>
<td>2.24</td>
</tr>
<tr>
<td>Tegra K1</td>
<td>12.5</td>
<td>0.8</td>
<td>6.2e-4</td>
<td>0.06</td>
<td>0.05</td>
</tr>
<tr>
<td>prototype</td>
<td>100</td>
<td>166</td>
<td>0.41</td>
<td>1.66</td>
<td>4.1</td>
</tr>
</tbody>
</table>

3.3.2 Multi-FPGA Design Space Exploration

Here we discuss the design space exploration of different mappings of the feed-forward stage of the CNN layer to an FPGA cluster.

3.3.2.1 Problem Formulation

In the deeply pipelined multi-FPGA architecture introduced in the previous section, CNN layers are implemented as compute engines on different FPGA boards. The ultimate goal is to find the best linear mapping of each CNN layer $i$ to $j$ to $k^{th}$ FPGA to form a pipeline of $K$ FPGAs, such that the overall latency is minimized or the overall throughput is maximized.

We assume that $K$, which is the total number of FPGAs in the pipeline cluster, is smaller than $N$ CNN layers.

The complexity of an exhaustive enumeration is exponential to the number of layers $N$ and FPGAs $K$, since the design space of the mappings are as many as $\binom{N-1}{K-1}$. Using this calculation, AlexNet on our 7-FPGA prototyping system can have up to 3003 different mappings.

In next part of this chapter we present a polynomial-time design space exploration solution using **dynamic programming**. The algorithm effectively reduces the problem of multi-FPGA design space exploration to a single-FPGA, which is solvable by extending existing solutions such as [ZLS15].

36
3.3.2.2 Optimized Multi-FPGA Mapping Solutions

The following equations present the solutions of throughput maximization over the whole design space, where their correctness can be proven by induction.

**Inter-Board Data Transfer Model** As discussed in Section 2.2.3, once a neural network is trained, its weights become constants. We assume that all weights are pre-stored in DRAM on corresponding FPGA boards during configuration time. So the data transferred between two FPGAs are the output feature maps of layer $i$, which is the last layer of the first FPGA. Then the data-transfer latency is:

$$T_{ext}(i) = \frac{D_i}{BW_{ext}} \quad (3.11)$$

where $D_i$ is the size of the output feature maps of layer $L_i$ and $BW_{ext}$ is the bandwidth of the board-to-board transmission.

**Latency minimization solution** The latency is measured as the time to process one image. We define $L(i, j, k)$ as the minimal latency of mapping layer $i$ to $j$ on $k$ FPGA. Therefore, the final solution of the latency-minimization mapping of a CNN model on up to $K$ FPGAs is:

$$\min_{k=1...K} L(1, N, k) \quad (3.12)$$

We can recursively compute $L(i, j, k)$ as the following:

$$L(i, j, k) = \begin{cases} 
L(i, j, 1), & k = 1 \\
\min_{r=i}^{j-1} \left( \begin{array}{c}
L(i, r, k - 1) + \\
L(r + 1, j, 1) + \\
T_{ext}(r)
\end{array} \right), & k \geq 2
\end{cases} \quad (3.13)$$

In the equation, $L(i, j, 1)$ is the minimum latency of implementing layer $i$ to layer $j$ on one FPGA, which can be obtained using existing work. In a CNN model with $N$ total layers,
there are $N \times (N + 1)/2$ different $L(i, j, 1)$. After these single-board solutions are computed, we need $O(N^2 \times K)$ computations to obtain the final result.

**Throughput maximization solution** The throughput can be measured as both the overall *giga-operations per second* (GOPS/s) or *images per second*. We define $T(i, j, k)$ as the maximal throughput of mapping layer $i$ to $j$ on $k$ FPGA. Therefore, the final solution of the *throughput maximization* mapping of a CNN model on up to $K$ FPGAs is:

$$\max_{k=1}^{K} T(1, N, k)$$

(3.14)

Since we also care about the overall energy efficiency, we define a metric of “throughput over power,” which will be GOPS/J or images/J. Assuming each FPGA board consumes $P$ Watt, the *energy efficiency maximization* solution is:

$$\max_{k=1}^{K} \frac{T(1, N, k)}{P \cdot k}$$

(3.15)

Both of these solutions can be obtained by recursively computing $T(i, j, k)$ as follows:

$$T(i, j, k) = \begin{cases} T(i, j, 1), & k = 1 \\ \max_{r=i}^{j-1} \min \left\{ \begin{array}{c} T(i, r, k - 1), \\ T(r + 1, j, 1), \\ \frac{1}{T_{\text{ext}}(r)} \end{array} \right\}, & k \geq 2 \end{cases}$$

(3.16)

In the equation, $T(i, j, 1)$ is the maximum throughput achieved by implementing layer $i$ to layer $j$ on one FPGA, which can be obtained using existing work. In a CNN model with $N$ total layers, there are $N \times (N + 1)/2$ different $T(i, j, 1)$s. After these single-board solutions are computed, we need $O(N^2 \times K)$ computations to obtain the result.

### 3.3.2.3 Single-FPGA Solution

In the previous sections, we presented a dynamic programming algorithm that reduces the problem of multi-FPGA design space exploration to the sub-problems of single-FPGA design.
space exploration. For the latency-minimization problem, the single-board optimization can be formulated as:

\[
L(i,j,1) = \min \sum_{r=i}^{j} L(r), \text{ s. t. } R \leq R_{FPGA}
\]

(3.17)

where \(L(r)\) is the compute time for layer \(r\), \(R\) is the total resource consumed by layer \(i\) to \(j\), and \(R_{FPGA}\) is the available resource on FPGA. For the throughput-maximization problem, the single-board optimization can be formulated similarly.

The execution cycle estimation for CONV, POOL and NL layers is formulated as an equation that relates the total number of arithmetic operations and loop unrolling factors in [ZLS15, QWY16]. Zhang et al. [ZLS15] proved that the uniformed configuration, in which two or more layers share the same hardware, only increases the layer-specific configuration within 5% in terms of execution cycles. Considering the frequency benefit from the reduced circuit complexity, the best uniformed configuration solution that is found by searching the whole design space already has the highest GOP/s performance. In this work we use a similar formulation.

**FC** layers can be viewed as a convolution [LSD15] with \(1 \times 1\) kernels on \(1 \times 1\) feature maps. Since the amount of weights is usually much larger than input feature maps, FC layers are memory-bound according to [QWY16]. In a max-throughput design, we choose to batch the input feature maps. Assuming a batch size of \(K\), then the FC computation becomes a convolution of \(1 \times 1\) kernel on \(K\) feature maps. Note that batching input feature maps does not improve the latency of each feature map.

**LRN** layers normalize input feature maps by each pixel with pixels from neighboring feature maps at the same position, whose computation pattern is described in Section 2.2.3. It outputs the same number of feature maps as that of the input. By deciding the unrolling factor for parallelizing multiple output feature maps, we are able to explore the trade-off between the execution cycles and resource cost. The total number of execution cycles are defined by
\[ \text{execution cycles} = \frac{\text{total arithmetic operations}}{\text{unroll factor}} \]
\[ = \frac{M \cdot R \cdot C \cdot (N + 4)}{U_{\text{lrn}}} \]  

(3.18)

where the notations follow equation 2.2. To compute the pixels from neighboring output feature maps from CONV, a stack of registers is used to buffer pixels from each of the CONV engine’s output BRAM buffers, which does shifting operations to feed LRN PEs with nearby pixels.

In summary, the unrolling factors of CONV and FCN layers are \(\langle T_m, T_n, T_k \rangle\), the POOL and NL layers unrolling factor is \(T_m\), and the LRN layer unrolling factor is \(T_l\), which defines each single-FPGA implementation. The total execution cycles of multiple layers on one single FPGA is the sum of the execution cycles of all layers.

### 3.3.3 Experimental Result

#### 3.3.3.1 Experiment Setup

The baseline system used in our evaluation is a workstation of an eight-core AMD A10-5800K CPU working at 3.8GHz, and a NVidia Titan X GPU [NVI15] plugged into the PCI-E slot. OpenBLAS and cuDNN library are used for software implementations.

For our prototype FPGA cluster (described in Section 3.3.1), we use Xilinx Vivado tools to synthesize FPGA bitstreams. The power consumption of both the baseline system and our prototype system is measured by a power meter.

#### 3.3.3.2 Evaluation of CNN Mappings

We first evaluate the mapping results under different objectives on the prototype system using two CNN models: AlexNet[KSH12] and VGG-16 [SZ14].

**Throughput and energy-efficiency maximization:** The throughput evaluations for AlexNet, based on different FPGA cluster sizes, are illustrated in Figure 3.6a and Figure 3.6c. For AlexNet, the best throughput is achieved at design A, which uses four FPGAs. Because
the throughout is flattened after four FPGAs, the best energy efficiency is also achieved in design A. For VGG-16, the overall throughput increases with more FPGA boards but the energy efficiency does not. Therefore, the most energy-efficient design for VGG is design C in Figure 3.6c using one FPGA, and design E achieves the highest throughput using six FPGAs.

Figure 3.6: Design space exploration of throughput (GOPS/s) and energy efficiency (GOPS/J) and latency (ms) maximization for AlexNet and VGG-16

Designs B and D achieve the best latency for VGG-16 and AlexNet respectively. Since in the latency-minimization design the FC layers are not batched, the total time is dominated by weight transfer. As a result, designs B and D achieve less energy efficiency compared to designs A and C, which are obtained from throughput-maximization solutions.

Observations: Several observations can be draw from Figure 3.6. First, the computation
pattern of AlexNet has more variation than VGG, especially since it contains a LRN layer while VGG does not. As shown in Table 3.6, LRN is very resource-costly, so it can become a system bottleneck when insufficient resource is allocated. This problem is illustrated in Figure 3.8. Second, the layer configuration of AlexNet has larger variations than VGG-16. Two out of five layers in AlexNet are followed by LRN layers, while VGG-16 has repeated CONV, NL and POOL layers. As a result, more FPGA resources in the pipeline help to get better acceleration for different layers. In addition, the convolution kernel sizes vary from $11 \times 11$ to $3 \times 3$ in AlexNet, which introduces more imbalance between the workload of different layers. Having more FPGA boards can improve workload balancing by allowing different configurations for each layer. In VGG-16, all layers use $3 \times 3$ kernels, so the imbalance issue
Figure 3.8: Performance comparison between CONV and LRN with different resource budgets. The horizontal axis ‘r’ denotes the ratio of DSP resource for CONV and LRN; higher ‘r’ means more DSPs are used for convolution and less for LRN. The vertical axis denotes execution cycles.

is minimal.

3.3.3.3 Overall System Performance and Energy Efficiency

Table 3.8 presents the comparisons between the designs generated on our FPGA cluster under different objectives and multiple baselines, including previous work on single-FPGA implementations.

For the designs generated from either throughput or energy optimization, the overall throughput is the highest among the previous single-FPGA accelerator designs for AlexNet and VGG. The energy efficiency is less than one previous work on Zynq [QWY16] because the Zynq board is for embedded systems, while our FPGA cluster incurs inevitable overheads like board-board communications and an additional FPGA as the master node (ZC706 in our case). However, we are still higher than previous throughput-oriented FPGA accelerator designs [SCD16]. The best throughput achieved by our prototype system is less than the GPU results, but it achieves $1.6 \times$ to $2 \times$ higher energy efficiency.

In all of our implementations, the top-1 and top-5 accuracy of the FPGA implementation of AlexNet and VGG are less than < 2% compared to the software implementation.
### Table 3.8: Comparison of CPU, GPU, FPGA implementations

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>AMD</td>
<td>NVIDIA</td>
<td>Zynq</td>
<td>Stratix-V</td>
<td>Virtex-7</td>
<td>Virtex-7</td>
<td>Virtex-7</td>
<td>Virtex-7</td>
<td>Virtex-7</td>
</tr>
<tr>
<td></td>
<td>A10</td>
<td>Titan X</td>
<td>XC7Z045</td>
<td>GSD8</td>
<td>VX690t</td>
<td>VX690t</td>
<td>VX690t</td>
<td>VX690t</td>
<td>VX690t</td>
</tr>
<tr>
<td>Technology</td>
<td>32nm</td>
<td>28nm</td>
<td>28nm</td>
<td>28nm</td>
<td>28nm</td>
<td>28nm</td>
<td>28nm</td>
<td>28nm</td>
<td>28nm</td>
</tr>
<tr>
<td>CNN Model</td>
<td>AlexNet</td>
<td>AlexNet</td>
<td>VGG &amp; AlexNet</td>
<td>VGG &amp; AlexNet</td>
<td>Alex</td>
<td>Alex</td>
<td>VGG</td>
<td>VGG</td>
<td>VGG</td>
</tr>
<tr>
<td>Precision</td>
<td>float</td>
<td>float</td>
<td>fixed(16b)</td>
<td>fixed(8-16b)</td>
<td>fixed(16b)</td>
<td>fixed(16b)</td>
<td>fixed(16b)</td>
<td>fixed(16b)</td>
<td>fixed(16b)</td>
</tr>
<tr>
<td>Accuracy Top-1</td>
<td>-</td>
<td>54.3%</td>
<td>68.02%</td>
<td>66.58%</td>
<td>52.4%</td>
<td>52.4%</td>
<td>66.51%</td>
<td>66.52%</td>
<td>66.51%</td>
</tr>
<tr>
<td>Accuracy Top-5</td>
<td>-</td>
<td>78.7%</td>
<td>87.94%</td>
<td>87.48%</td>
<td>77.83%</td>
<td>77.83%</td>
<td>86.89%</td>
<td>86.92%</td>
<td>90.88%</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>3,800</td>
<td>∼ 1,000</td>
<td>150</td>
<td>120</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>Power (Watt)</td>
<td>87.3</td>
<td>328.3</td>
<td>9</td>
<td>19.1</td>
<td>126</td>
<td>126</td>
<td>35</td>
<td>35</td>
<td>160</td>
</tr>
<tr>
<td>Batch Size</td>
<td>16</td>
<td>256</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td># of FPGAs</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Through. (GOPS)</td>
<td>34.23</td>
<td>1385.5</td>
<td>137.0</td>
<td>117.8</td>
<td>825.6</td>
<td>128.8</td>
<td>290</td>
<td>203.9</td>
<td>1280.3</td>
</tr>
<tr>
<td>Latency (ms)</td>
<td>83.6</td>
<td>89.7</td>
<td>224.6</td>
<td>262.9</td>
<td>104</td>
<td>30.6</td>
<td>213.6</td>
<td>151.8</td>
<td>200.9</td>
</tr>
<tr>
<td>E.-E.(GOPS/J)</td>
<td>0.39</td>
<td>4.22</td>
<td>15.2</td>
<td>6.17</td>
<td>6.55†</td>
<td>1.02†</td>
<td>8.28†</td>
<td>5.83†</td>
<td>8.00†</td>
</tr>
</tbody>
</table>

* Optimization objective for 'Energy' uses the metric of 'GOPS/J' and objective for 'Latency' uses 'second/image.'
† 1+N represents 1 Zynq board plus N VC709 boards.
‡ The power consumption also include the ZC706 board; therefore the overall energy-efficiency is less than reported in Figure 3.6.
3.4 Conclusions

In this chapter we first describe a platform-based methodology for the domain of neuron simulation. We discuss the hierarchical structure of neural microcircuits and demonstrate the mapping to digital logic using a high-level synthesis approach. Using the neural microcircuits related to oscillatory path integration as a case study, we show that our platform is significantly faster and more energy-efficient than general-purpose processors. Our platform enables domain experts to create efficient and scalable FPGA simulation engines without in-depth knowledge of hardware design. For hardware design experts, our platform also provides opportunities to conduct design space exploration with high-level synthesis tools to further optimize performance.

Similar techniques can be applied to neural networks as well. Like the populations in a neural microcircuit, modern deep convolutional neural networks are composed of different layers. Different layers can be mapped to one or multiple compute engines, with the data streaming through different layers. This design methodology is demonstrated with a deeply pipelined multi-FPGA architecture. Various strategies are applied to optimize CNN mapping to multi-FPGA platforms. We propose a dynamic programming method to efficiently explore the design space for both efficiency and latency. We build a prototype of six FPGAs to verify our idea. With two famous CNN models as case studies, we demonstrate that our prototype can achieve up to 21× and 2× energy-efficiency compared to optimized multi-core CPU and GPU implementations, respectively. The overall throughput and latency results of the prototype system are also better than existing single-FPGA implementations. The proposed architecture can also be easily integrated with the latest single-board FPGA design to achieve better energy-efficiency.
CHAPTER 4

Runtime System for Large-Scale Accelerator Deployment

In the previous chapter, the focus is on single-node accelerator system design. With big-data analytics growing into an unprecedented scale, customized hardware acceleration needs to be deployed at datacenter-scale to improve the energy-efficiency and sustain datacenter scaling. In this next chapter, we discuss the acceleration deployment focusing on the scale-out design methodology.

4.1 Prototyping Scale-out Systems and Evaluation

In the beginning of this chapter, we explore the design options in heterogeneous datacenters with FPGA accelerators with quantitative studies on a wide range of systems, including an Xeon cluster, an Xeon cluster with FPGA accelerator attached to the PCI-E bus, a low-power Atom CPU cluster, and a cluster of embedded ARM processors with on-chip FPGA accelerators. The following observations can be drawn from the experiments:

Observation 1: By experimenting with different distributed machine learning workloads on an Intel Atom cluster and an ARM cluster, we conclude that although the small-core CPU clusters consume only 0.2× to 0.3× of the power of a server-class cluster, the performance may slow down by as much as 10×. As a result, the total energy consumed by these low-power processors is still many times more than their server-class counterparts.

Observation 2: Our experiments demonstrate that poor performance of small-core CPUs can be compensated for by accelerators built on the same chip. Our prototyping
cluster composed from Xilinx Zynq SoCs [RBD11], which have both embedded ARM cores and FPGA fabrics, demonstrated 8× to 15× speedup and energy reduction compared to Atom and ARM systems, and 2x performance gain and energy reduction compared to a regular Xeon server.

**Observation 3:** The FPGA accelerator is more effective for big-core systems compared to small-core systems for big-data analytic applications. One of our experiments shows that the performance of one Xeon server plus FPGA accelerator is around 2× faster than eight nodes of Zynq, even though the aggregated computing power of the FPGA fabrics on these eight nodes is around 2× more than the FPGA in the Xeon server. The main reason is that on the Zynq cluster, the sequential part of the program becomes more dominant after acceleration, and the cost of moving data to the accelerator is higher. In terms of energy efficiency, on the other hand, the difference between the two types of systems is very small.

To evaluate the performance and energy efficiency of various accelerator-rich systems, several real prototype hardware systems are built to experiment with real-world big-data applications.

### 4.1.1 Prototyping System Design

#### 4.1.1.1 Baseline Big-Core and Small-Core Systems

For the baseline of big-core CPU systems, we built a cluster with dual-core Intel Xeon CPU servers connected with both 1G and 10G Ethernet. The cluster contains more than 20 server nodes. A snapshot of the cluster is shown in Figure 4.1.

For the baseline of small-core CPU systems, we used a cluster of eight nodes of Intel Atom CPUs and a cluster of eight nodes of embedded ARM cores. The ARM cluster is the same as our prototype presented later in this section.
4.1.1.2 Big-Core with PCIE Accelerators

Similar to existing GPGPU platforms, FPGA accelerators can also be integrated into normal server nodes with PCIE slots. Taking advantage of the energy efficiency of the FPGA chips, these PCIE accelerator boards do not require an external power supply, which makes it possible to deploy FPGA accelerators into datacenters without the need to modify existing infrastructures.

In our experiments, we integrate AlphaData (AD) FPGA boards into our Xeon cluster in Figure 4.1. Each FPGA board contains a Xilinx Virtex-7 XC7VX690T-2 FPGA chip with 16GB of on-board memory.

4.1.1.3 Small-Core with On-Chip Accelerators

We also built a customized cluster of low-power CPU cores with on-chip FPGA accelerator. The Xilinx Zynq SoC was selected as the experimental heterogeneous SoC, which includes a processing system based on dual ARM A9 cores and a programmable FPGA logic. The accelerators are instantiated on the FPGA logic and can be reconfigured during runtime. We build a cluster of eight Zynq nodes.

The entire hardware system is built with off-the-shelf commodity hardware. A snapshot
The system overview of the prototype cluster of the system is shown in Figure 4.2b. Each node in the cluster is a Xilinx ZC706 board, which contains a Xilinx Zynq XC7Z045 chip. Each board also has 1GB of on-board DRAM and a 128GB SD card used as a hard disk. The ARM processor in the Zynq SoC shares the same DRAM controller as well as address space with the programmable fabrics. The processor can control the accelerators on the FPGA fabrics using two system buses. The memory is shared through four high-performance memory buses (HPs) and one coherent memory bus (ACP). All the boards are connected to a Gigabit Ethernet switch. The hardware layout of the Zynq boards and their connection is shown in Figure 4.2a in the bottom box for the ZC706 board.

The software setup and accelerator integration method are shown in the upper box in Fig. 4.2a. A lightweight Linux system is running on the ARM processors of each Zynq board, which provides drivers for peripheral devices such as Ethernet and SD card, and also controls the on-chip FPGA fabrics. To instantiate our machine learning accelerators on the FPGA, we design a driver module to configure the control registers of the accelerators as memory-mapped IOs, and use DMA buffers to facilitate data transfers between the host system and the accelerators. The machine learning accelerators are synthesized as FPGA configuration bitstreams and can be programmed on the FPGA at runtime.
4.1.1.4 System Profile Summary

With our prototype clusters, we can evaluate heterogeneous accelerator-rich systems on different scales. A summary of the specifications and configurations of CPU and FPGA architectures that are used in our prototype platforms are presented in Table 4.1 and Table 4.2, respectively.

Table 4.1: Prototype System Specifications

<table>
<thead>
<tr>
<th>Item</th>
<th>Model</th>
<th>Frequency</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon</td>
<td>E5-2620v3</td>
<td>2.4GHz</td>
<td>22nm</td>
</tr>
<tr>
<td>Atom</td>
<td>D2500</td>
<td>1.86GHz</td>
<td>32nm</td>
</tr>
<tr>
<td>Zynq-ARM</td>
<td>Cortex A9</td>
<td>800MHz</td>
<td>28nm</td>
</tr>
<tr>
<td>Zynq-FPGA</td>
<td>XC7Z045</td>
<td>200MHz</td>
<td>28nm</td>
</tr>
<tr>
<td>AD-FPGA</td>
<td>XC7VX690T-2</td>
<td>200MHz</td>
<td>28nm</td>
</tr>
</tbody>
</table>

Table 4.2: Experimental Platform Configurations

<table>
<thead>
<tr>
<th>Item</th>
<th>Configuration</th>
<th>Avg. Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-Core</td>
<td>Xeon</td>
<td>175W</td>
</tr>
<tr>
<td>Small-Core</td>
<td>Atom</td>
<td>30W</td>
</tr>
<tr>
<td>Smaller-Core</td>
<td>Zynq ARM</td>
<td>10W</td>
</tr>
<tr>
<td>Smaller-Core+FPGA</td>
<td>Zynq</td>
<td>10W</td>
</tr>
<tr>
<td>Big-Core+FPGA</td>
<td>Xeon + AD-FPGA</td>
<td>200W</td>
</tr>
</tbody>
</table>

4.1.2 Experimental Results

4.1.2.1 Application Case Studies and Accelerator Designs

Today, machine learning has become the core of the big-data applications. The huge success of Internet services such as data-mining, web-search and advertisement drives the rapid
development of machine learning algorithms. In our evaluation, we select two widely used machine learning algorithms in the experiments: logistic regression (LR) and K-Means clustering (KM).

**Logistic Regression (LR):** Logistic regression [Fre09] is a combination of a linear model and a logistic function, which regulates the output between 0 and 1. The baseline of LR in our experiments is the training application implemented by Spark MLlib, with the LBFGS algorithm.

**K-Means clustering (KM):** K-Means clustering is an iterative algorithm which classifies data points (features) into several clusters. The baseline KM implementation in our experiments is also from Spark MLlib. The compute kernel selected is the local sum of center distances calculation. The datasets used in K-Means are the same as LR.

Both of these applications are iterative, so that they can take advantage of the data caching capability provided by the Blaze runtime system to mitigate the data transfer overheads between the host Spark program and FPGA accelerators. The input data set of the MNIST handwritten digits [LC98] is selected for both LR and KM.

### 4.1.2.2 Accelerator Kernel Design

The LR and KM accelerator kernels used in our experiments are from the Machine Learning FPGA Acceleration Library from Falcon Computing Solutions [fcs]. They are written as parameterized C++ code for Xilinx Vivado HLS. For all the designs on the AlphaData (AD) FPGA board, we use the Xilinx SDAccel which automatically generates the FPGA bitstream and exposes an interface to OpenCL. For Zynq designs, we directly use the Xilinx Vivado toolchain to get the bitstream.

Table 4.3 summarizes the resource consumption of our accelerator designs on different devices, as well as the speedup compared to single-core performance of the Xeon CPU in our cluster.
Table 4.3: Specifications of our FPGA accelerators

<table>
<thead>
<tr>
<th>App</th>
<th>Device</th>
<th>LUTs</th>
<th>FF</th>
<th>DSP</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>AD</td>
<td>54%</td>
<td>37%</td>
<td>45%</td>
<td>42x</td>
</tr>
<tr>
<td>LR</td>
<td>Zynq</td>
<td>58%</td>
<td>28%</td>
<td>48%</td>
<td>11x</td>
</tr>
<tr>
<td>KM</td>
<td>AD</td>
<td>34%</td>
<td>18%</td>
<td>20%</td>
<td>26x</td>
</tr>
<tr>
<td>KM</td>
<td>Zynq</td>
<td>22%</td>
<td>6%</td>
<td>18%</td>
<td>7x</td>
</tr>
</tbody>
</table>

4.1.2.3 System Performance and Energy Results

Experiment Methodology  In the experiments discussed in this section, we measure the total application time, including the initial data load and communication. The energy consumption is calculated by measuring the average power consumption during operation using a power meter and multiplying it by the execution time, since we did not observe significant variations of system power during our experiments. All the energy consumption measurements also include a 24-port 1G Ethernet switch.

Big-Core vs. Big-Core + FPGA  We first present the effectiveness of FPGA accelerators in a common datacenter setup. Fig. 4.3 includes the comparison between a CPU-only cluster and a cluster of CPU with PCIE FPGA boards using LR, KM. For the machine learning workloads where most of the computation can be accelerated, FPGA can contribute to significant speedup with only a small amount of extra power. More specifically, the big-core plus FPGA configuration achieves $3.05 \times$ and $1.47 \times$ speedup for LR and KM respectively and reduces the overall energy consumption to 38% and 56% of the baseline respectively.

Small-Core + FPGA vs. Big-Core + FPGA  We then evaluate the performance and energy consumption between big-core with FPGA and small-core with FPGA with the application of LR and KM. Fig. 4.3 illustrates the execution time and energy consumption of running LR and KM applications on different systems. Notably, both the performance and energy efficiency of pure Atom and ARM clusters are worse than the single Xeon server,
Figure 4.3: Execution time (above) and energy consumption (below) normalized to the results on one Xeon server.

which confirms the argument in [KRD12] that low-power cores could be less energy-efficient for computation-intensive workloads.

Several observations can be drawn from the results in Figure 4.3. First, for both small-core and big-core systems, the FPGA accelerators provide significant performance and energy-efficiency improvement—not only for kernels but also for the entire application. Second, compared to big-core systems, small-core systems benefit more from FPGA accelerators. This means that it is more crucial to provide accelerator support for future small-core-based datacenters. Finally, although the kernel performance on eight Zynq FPGAs is better than one AD FPGA, the application performance of Xeon with AD-FPGA is still 2× better than Zynq. This is because on Zynq the non-acceleratable part of the program, such as disk I/O and data copy, is much slower than Xeon. On the other hand, the difference in energy-efficiency between Xeon plus FPGA and Zynq is much smaller.

4.1.2.4 Summary of Different Systems

Finally, Table 4.4 summarizes our evaluation of different heterogeneous platforms. For our application case study of compute-intensive machine learning applications, pure small-core systems have the worst performance and consume the most energy. This means in future
datacenters, it can be unrealistic to have a small-core-only configuration. Small-core plus FPGA configurations, on the other hand, have the best energy-efficiency and can also provide better performance than big-core-only configurations.

Table 4.4: Summary of experimental results on different platform configurations

<table>
<thead>
<tr>
<th>Item</th>
<th>Performance</th>
<th>Energy-Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-Core + FPGA</td>
<td>Best</td>
<td>Best</td>
</tr>
<tr>
<td>Small-Core + FPGA</td>
<td>Better</td>
<td>Best</td>
</tr>
<tr>
<td>Big-Core only</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>Small-Core only</td>
<td>Bad</td>
<td>Bad</td>
</tr>
</tbody>
</table>

4.2 Blaze Runtime System

Among various solutions that harness accelerators in a datacenter, the FPGA-enabled datacenter has gained increased attention and is considered one of the most promising approaches. This is because FPGAs provide low power, high energy efficiency, and reprogrammability to customize high-performance accelerators. One breakthrough example is that Microsoft has deployed FPGAs into its datacenters to accelerate the Bing search engine with almost 2x throughput improvement while consuming only 10% more power per CPU-FPGA server [Put14]. Another example is IBM’s deployment of FPGAs in its data engine for large NoSQL data stores [BRH15]. Moreover, Intel, with the $16.7 billion acquisition of Altera, is providing closely integrated CPU-FPGA platforms for datacenters [harb], and is targeting the production of around 30% of the servers with FPGAs in datacenters by 2020 [hara]. Following this trend, Amazon Web Service (AWS) has announced their FPGA-enabled compute instance F1 in their EC2 offering in February 2017 [ama].

With the emerging trend of FPGA-enabled datacenters, one key question is: How can we easily and efficiently deploy FPGA accelerators into state-of-the-art big-data computing systems like Apache Spark [ZCF10] and Hadoop YARN [VMD13]? To achieve this goal, both programming abstractions and runtime support are needed to make these existing systems
programmable to FPGA accelerators. This is challenging for the following reasons.

1. Unlike conventional CPU and GPU targeted programs, compiling an FPGA program can take several hours, which makes existing runtime systems that use dynamic code generation for CPU-GPU datacenters, such as Dandelion [RYC13], HadoopCL [GBS13] and SWAT [GS16], not applicable for FPGAs.

2. State-of-the-art big-data systems like Apache Hadoop and Spark compile to the Java Virtual Machine (JVM), while FPGA accelerators are usually manipulated by C/C++/OpenCL. Even with predesigned FPGA accelerators, there are still excessive programming efforts required to i) integrate them with the JVM, ii) share an accelerator among multiple threads or multiple applications, and iii) share an FPGA platform by multiple accelerators of different functionalities.

3. A straightforward JNI (Java Native Interface) integration of FPGA accelerators can diminish or even degrade the overall performance (up to 1000X slowdown) due to the overwhelming JVM-to-native-to-FPGA communication overhead [CCF16].

4. It usually takes several seconds to reprogram an FPGA into a different accelerator (with a different functionality). A frequent FPGA reprogramming in a multi-accelerator scenario can significantly degrade the overall system performance. This raises a fundamental question: Do we manage “the hardware platform itself” or “the logical accelerator (functionality) running on top of the hardware platform” as a resource?

To address these challenges, we design and implement Blaze: a framework that provides a programming abstraction and runtime support for easy and efficient FPGA deployments in datacenters. Blaze has the following contributions.

1. Programming APIs that enable big-data processing applications to leverage FPGA accelerators to perform task-level work. We abstract FPGA accelerators as a service (FaaS), which decouples the hardware accelerator development of data processing tasks (i.e., Spark transformations) and big-data processing logic (i.e., scheduling tasks, shuffling data, etc.).
2. Policies for managing logical accelerator functionality—instead of the physical hardware platform itself—as a resource, where better scheduling decisions can be made to optimize the system throughput and energy efficiency.

3. An efficient runtime to share FPGA accelerators in data-centers, where an FaaS framework is implemented to support sharing of accelerators among multiple threads and multiple applications in a single node. Also, an accelerator-centric scheduling is proposed for the global accelerator management to alleviate the FPGA reprogramming overhead for multi-accelerators. Finally several well-known optimization techniques—such as data caching and task pipelining—are employed to reduce the JVM-to-FPGA communication overhead.

4. An open-source prototype that is compatible with existing ecosystems like Apache Spark with no code changes and YARN with a lightweight patch. Our goal is to bring FPGA accelerator developers, big-data application developers, and system architects together, to blaze the deployment of accelerators in datacenters.  

4.2.1 Blaze System Overview

We design Blaze as a generic system to enable big-data applications to easily access FPGA accelerators and implement it as a third-party package that works with existing ecosystems (i.e., Apache Spark and Hadoop YARN), with lightweight changes. Here we give an overview of the Blaze programming and runtime support and discuss how we address the challenges mentioned before.

To provide an easy-to-use programming interface, we abstract FPGA accelerators as a service (FaaS) and propose to decouple the software development of big-data applications and the hardware development of FPGA accelerators. This means hardware experts can make the best effort to optimize the accelerator design without being burdened with application

---

1Blaze can be downloaded from github: https://github.com/UCLA-VAST/blaze. Blaze has already been used by multiple groups at Intel Labs to deploy accelerators composed of the Intel-Altera Heterogeneous Accelerator Research Platforms (HARP CPU-FPGA platforms).
complexity, and software developers do not need to be aware of tedious hardware details to take advantage of accelerators. Currently, Blaze provides a set of APIs for Spark programs to offload map computations onto accelerators without any change to the Spark framework. All Spark programmers have to do is to register the pre-defined FPGA accelerators (developed by hardware experts) into Blaze as a service, and call the Blaze API to access the customized accelerators. All the accelerator sharing and management logic are transparently handled by our Blaze runtime.

The Blaze runtime system integrates with Hadoop YARN to manage accelerator sharing among multiple applications. As illustrated in Figure 4.4, Blaze includes two levels of accelerator management. A global accelerator manager (GAM) oversees all the accelerator resources in the cluster and distributes them to various user applications. Node accelerator managers (NAMs) sit on each cluster node and provide transparent accelerator access to a number of heterogeneous threads from multiple applications. After receiving the accelerator computing resources from GAM, the Spark application begins to offload computation to the accelerators through NAM. NAM monitors the accelerator status, handles JVM-to-FPGA data movement and accelerator task scheduling. NAM also performs a heartbeat protocol with GAM to report the latest accelerator status.

We summarize the key features of Blaze as follows.

1. **FPGA accelerators as a service (FaaS).** The most important role of NAM in Blaze runtime is providing transparent FaaS shared by multiple application jobs (run on the same node) that request accelerators in a fashion similar to software library routines. Each “logical accelerator” library routine exposes a predefined functionality to a Spark program, and can be composed of multiple “physical accelerators” on multiple hardware platforms (e.g., two FPGAs, or one FPGA and one GPU). FaaS automatically manages the task scheduling between logical and physical accelerators. For example, multiple physical accelerators can be allocated for a single logical accelerator for performance-demanding applications, while one physical accelerator can be shared across multiple logical accelerators if each has a low utilization of that physical
2. **Accelerator-centric scheduling.** In order to solve the global application placement problem considering the overwhelming FPGA reprogramming overhead, we propose to manage the logical accelerator functionality, instead of the physical hardware itself, as a resource to reduce such reprogramming overhead. We extend the *label-based scheduling* mechanism in YARN to achieve this goal: instead of configuring node labels as ‘FPGA’, we propose to use accelerator functionality (e.g., ‘KMeans-FPGA’, ‘Compression-FPGA’) as node labels. This helps us to differentiate applications that are using the FPGA devices to perform different computations. Therefore, we can delay the scheduling of accelerators with different functionalities onto the same FPGA to avoid reprogramming as much as possible. Different from the current YARN solution, where node labels are configured into YARN’s configuration files, node labels in Blaze are configured into NAM through command-line. NAM then reports the accelerator information to GAM through heartbeats, and GAM configures these labels into YARN.

3. **Hiding JVM-to-FPGA communication.** We also employ well-known techniques
such as data caching and task pipelining in FaaS to hide the overwhelming JVM-to-native-to-FPGA communication overhead.

4. **Fault tolerance.** The FaaS design in each NAM also helps the fault tolerance of the system. Whenever a fault in the accelerator hardware occurs, NAM can allocate different hardware to fulfill the request, or fallback to CPU execution when no more accelerators are available.

5. **Facilitating rolling upgrades.** FaaS makes it easy to configure heterogeneous accelerator resources on compute nodes in the datacenter, facilitating rolling upgrades of next-generation accelerator hardware and making the system administration of large-scale heterogeneous datacenters more scalable.

In summary, the easy-to-use programming interface, transparent FaaS, and the accelerator-centric scheduling of Blaze makes FPGA accelerator deployment at datacenter scale much easier than existing approaches. Note that the FaaS framework for NAM is provided as a third-party package without any change to Apache Spark, while accelerator-centric scheduling for GAM and NAM is provided as a lightweight patch to Hadoop YARN. In Section 4.2.2 and Section 4.2.3, we will present more details about the Blaze programming interface and runtime implementation.

### 4.2.2 Blaze Programming Interface

We first describe the programming interfaces of Blaze from two aspects: how to write a big-data application that invokes FPGA accelerators, and how to design and register an FPGA accelerator into Blaze. Then we present our support for data serialization during data transfer between JVM and accelerators.
4.2.2.1 Application Programming Interface

We implement Blaze as a third-party package that works with the existing Spark framework without any modification of Spark source code. Thus, Blaze is not specific to a particular version of Spark. Moreover, the Blaze programming model for user applications is designed to support accelerators with minimal code changes. To achieve this, we extend the Spark RDD to AccRDD which supports accelerated transformations. We explain the detailed usage of AccRDD in Listing 4.1 with an example of logistic regression.

Listing 4.1: Blaze application example (Spark Scala)

```scala
val points = sc.textFile(filePath).cache()
val train = blaze.wrap(points)
for (i <- 0 until ITERATIONS) {
  bcW = sc.broadcast(weights)
  val gradients = train.map(
    new LogisticAcc(bcW)
  ).reduce(a + b)
  weights -= gradients
}
```

```scala
class LogisticAcc(w: Broadcast_var[V])
  extends Accelerator[T, U] {
    val id: String = "LRGradientCompute"
    def call(p: T): U = {
      localGradients.compute(p, w.value)
    }
  }
```

In Listing 4.1, training data samples are loaded from a file and stored to an RDD `points`, and are used to train `weights` by calculating gradients in each iteration. To accelerate the gradient calculation with Blaze, first the RDD `points` needs to be extended to AccRDD `train` by calling the Blaze API `wrap`. Then an accelerator function, `LogisticAcc`, can be passed to the `.map` transformation of the AccRDD. This accelerator function is extended from

---

2Blaze also supports C++ applications with similar interfaces, but we will mainly focus on Spark applications in this thesis.
the Blaze interface **Accelerator** by specifying an accelerator id and an optional **compute** function for the fall-back CPU execution. The accelerator id specifies the desired accelerator service, which in the example is “LRGradientCompute”. The fall-back CPU function will be called when the accelerator service is not available. This interface is provided with fault-tolerance and portability considerations. In addition, Blaze also supports caching for Spark broadcast variables to reduce JVM-to-FPGA data transfer. This will be elaborated in Section 4.2.3.2.

The application interface of Blaze can be used by library developers as well. For example, Spark MLlib developers can include Blaze-compatible codes to provide acceleration capabilities to end users. With Blaze, such capabilities are independent of the execution platform. When accelerators are not available, the same computation will be performed on CPU. In this case, accelerators will be totally transparent to the end users. In our evaluation, we created several implementations for Spark MLlib algorithms such as logistic regression and K-Means using this approach.

### 4.2.2.2 Accelerator Programming Interface

For accelerator designers, the programming experience is decoupled with any application-specific details. An example of the interface implementing the “LRGradientCompute” accelerator is shown in Listing 4.2.

Our accelerator interface hides details of FPGA accelerator initialization and data transfer by providing a set of APIs. In this implementation, for example, the user inherits the provided template, **Task**, and the input and output data can be obtained by simply calling **getInput** and **getOutput** APIs. No explicitly OpenCL buffer manipulation is necessary for users. The runtime system will prepare the input data and schedule it to the corresponding task. The accelerator designer can use any available programming framework to implement an accelerator task as long as it can be integrated with an interface in C++.

Listing 4.2: Blaze accelerator example (C++)

```cpp
class LogisticTask : public Task {

```
4.2.2.3 Serialization Support

The input and output data of Spark tasks need to be serialized and deserialized respectively before they are transferred to and from accelerator platforms. Blaze implementation includes its own (de)serializer for primitive data types, because the existing Java version is not sufficient for handling the data layout for accelerators. In addition, Blaze also provides an interface to users to implement their own (de)serializer methods. As a result, users are allowed to use arbitrary data types in the Spark application as long as the corresponding (de)serializer is able to process data to match the accelerator interface.

4.2.3 Blaze Runtime Support

In this section, we present our Blaze runtime support, including the FaaS implementation to share accelerators among multiple heterogeneous threads in a single node, accelerator-centric scheduling to alleviate the FPGA reprogramming overhead, communication optimization to alleviate the JVM-to-FPGA overhead, and fault tolerance and security support.
Figure 4.5: Node accelerator manager design to enable FPGA accelerators as a service (FaaS).

4.2.3.1 FPGA-as-a-Service (FaaS)

Blaze facilitates FaaS in NAM through two levels of queues: task queues and platform queues. The architecture of NAM is illustrated in Figure 4.5. Each task queue is associated with a “logical accelerator”, which represents an accelerator library routine. When an application task requests a specific accelerator routine, the request is put into the corresponding task queue. Each platform queue is associated with a “physical accelerator”, which represents an accelerator hardware platform such as an FPGA board. The tasks in task queue can be executed by different platform queues depending on the availability of the implementations. For example, if both GPU and FPGA implementations of the same accelerator library routine are available, the task of that routine can be executed on both devices.

This mechanism is designed with three considerations: 1) application-level accelerator sharing, 2) minimizing FPGA reprogramming, and 3) efficient overlapping of data transfer
and accelerator execution to alleviate JVM-to-FPGA overhead.

In Blaze, accelerator devices are owned by NAM rather than individual applications. The reasoning behind this design is our observations that in most big-data applications, the accelerator utilization is less than 50%. If the accelerator is owned by a specific application, then much of the time it will be spent in idle, wasting energy. The application-level sharing inside NAM is managed by a scheduler that sits between application requests and task queues. In this thesis, a simple first-come-first-serve scheduling policy is implemented. We leave the exploration of different policies to future work.

The downside of providing application sharing is the additional overheads of data transfer between the application process and NAM process. For latency-sensitive applications, Blaze also offers a reservation mode where the accelerator device is reserved for a single application, i.e., a NAM instance will be launched inside the application process.

The design of the platform queue focuses on mitigating the large overhead in FPGA reprogramming. For a processor-based accelerator such as GPU to begin executing a different “logical accelerator”, it simply means loading another program binary, which incurs minimum overhead. With FPGA, on the other hand, the reprogramming takes much longer. An FPGA device contains an array of logic cells, and the programming is effectively configuring the logic function and connection of each cell. Each configuration is called a “bitstream”, and it typically takes 1∼2 seconds to program an FPGA with a given bitstream. Such a reprogramming overhead makes it impractical to use the same scheme as the GPU in the runtime system. In Blaze, a second scheduler sits between task queues and platform queues to avoid frequent reprogramming of the same FPGA device.

4.2.3.2 Hiding JVM-to-FPGA Communication

In order for a Spark program to transfer data to an FPGA accelerator, the data has to be first moved from JVM to the native machine, and then moved to the FPGA device memory through a PCIe connection. Such data movement between the host CPU and FPGA accelerators sometimes can diminish or even degrade the overall system performance [CCF16]. To
mitigate such overhead, Blaze adopts the following well-known techniques within the FaaS framework.

1. **Task pipelining.** Most datacenter workloads will have multiple threads/tasks sharing the same accelerator, which creates an opportunity to hide data transfer with task execution by pipelining: the task queue in NAM adopts an asynchronous communication scheme that overlaps JVM-to-FPGA data communication with FPGA accelerator execution.

2. **FPGA data caching.** Many big-data applications like machine learning use iterative algorithms that repeatedly perform computation on the same set of input data. This provides the opportunity to cache the data on the FPGA device memory and thus avoid the most time-consuming native-to-FPGA data movement through PCIe. To be more specific, our FaaS framework implements a Block Manager to maintain a data reuse table that records the mapping from the native data block to the FPGA device memory block. For the case of OpenCL, Block Manager manages a table of cl_buffer objects which are mapped to device memory. A flag is used to indicate whether the programmer wants Blaze to cache an input data block. In Spark, the flag is automatically assigned if the user specifies `.cache()` for the input RDD.

3. **Broadcast data caching.** Most data analytic frameworks such as Spark support data sharing across the cluster nodes. In Spark, this is provided as broadcast data. Similarly, Blaze also supports a broadcast data caching to minimize data transfer across the cluster nodes. A broadcast block only needs to be transferred to the NAM once, and it will be cached inside the Block Manager throughout the application’s life cycle.

4.2.3.3 Fault Tolerance and Security Issues

Fault tolerance is inherent in our proposed transparent accelerator scheduling. All accelerator-related errors are caught at the application level, and the CPU implementation will be used to resume the execution. Errors of accelerators in NAM are handled in a similar fashion as
Spark or YARN. A counter is used for each accelerator task per platform, keeping track of
the number of errors incurred. If the failure is persistent for one accelerator task, it will be
removed from NAM’s configuration. This information will also be propagated to GAM in
the heartbeat signals, and GAM will remove the corresponding label for this node.

Based on the description of the Blaze accelerator interface in Section 4.2.2.2, the accel-
erator task implementation only has access to its private input data through the provided
interface, such as `getInput()`. The data can only be assigned by NAM based on the de-
pendency, and all input data is read-only. Our underlying platform implementation is based
on existing accelerator runtime systems such as OpenCL, so we rely on the runtime im-
plementation to guarantee security at the device level. In general, the security issues in
FPGA-enabled datacenters will be an open and interesting

4.2.4 Experimental Results

In this section, we evaluate the programming efforts and system performance of deploying
FPGA accelerators in datacenters using Blaze. First we present the hardware and software
setup, and describe the four representative large-scale applications we chose that cover two
extremes: iterative algorithms like machine learning, and streaming algorithms like compres-
sion and genome sequencing. We evaluate the programming efforts to write these applications
using Blaze in terms of lines-of-code (LOC). Then we evaluate the overall system speedup
and energy savings for each individual application by putting FPGA accelerators into the
cluster. We also analyze the FaaS overhead and break down the performance improvement
of each optimization. Finally, we analyze multi-job executions and the efficiency of our
accelerator-centric scheduling policy in the global accelerator management.

4.2.4.1 Experimental Setup

The experimental platform we use is a local standard CPU cluster with up to 20 nodes, among
which 4 nodes are integrated with FPGA cards using PCI-E slots. Each server has dual-
socket Intel Xeon E5-2620v3 CPUs with 12 cores in total and 64GB of main memory. The
FPGA card is AlphaData ADM-PCIE-7V3, which contains a Xilinx Virtex-7 XC7VX690T-2 FPGA chip and 16GB of on-board DDR3 memory. The FPGA board can be powered by PCI-E alone and consumes around 25W, which makes it deployable into commodity datacenters.

The software framework is based on a community version of Spark 1.5.1 and Hadoop 2.6.0. The accelerator compilation and runtime are provided by the vendor toolkits. For the AlphaData FPGA cards, we use the OpenCL flow provided by the Xilinx SDAccel tool-chain, where the OpenCL kernels will be synthesized into bitstreams to program the FPGA.

We choose a set of four representative compute-intensive large-scale applications. They cover two extremes: iterative machine learning algorithms like logistic regression and K-means clustering, and streaming algorithms like genome sequencing analysis and Apache Parquet compression.

1. **Logistic regression (LR)**. The baseline LR is the training application implemented by Spark MLlib [mll] with the LBFGS algorithm. The software baseline uses netlib with native BLAS library. The computation kernels we select are the logistic gradients and the loss function calculation. The kernel computation takes about 80% of the total application time.

2. **K-Means clustering (KM)**. The KM application is also implemented using Spark MLlib, which uses netlib with native BLAS library. The computation kernel we select is the local sum of center distances calculation. The datasets used in KM are the same as LR, and the percentage of kernel computation time is also similar to LR.

3. **Genome sequences alignment (GSA)**. The GSA application is from the open-source Cloud Scale BWAMEM (CS-BWAMEM) software suite [CCL15c], which is a scale-out implementation of the BWAMEM algorithm [Li13a] widely used in the bioinformatics area. The algorithm aligns the short reads from the sequencer to a reference genome. We mainly focus on the alignment step in this application which uses the Smith-Waterman algorithm, as we did in a prior case study [CCF16].
4. **Apache Parquet compression (COMP)**. Apache Parquet [par] is a compressed and efficient columnar data representation available to any project in the Hadoop/Spark ecosystem. Such columnar data generally have good compression rates and thus are often compressed for better spatial utilization and less data communication. We mainly focus on the compression (deflater) step, which is computation-bound and common through various applications. We use two software baselines: 1) the Java Gzip implementation that uses both the LZ77 algorithm and Huffman encoding, which has a better compression ratio but low throughput; and 2) the open-source Snappy implementation [sna] that uses a JNI wrapper to call the C++ Snappy library based on the LZ77 algorithm, which has a lower compression ratio but better throughput.

The input data for LR and KM are based on a variant of the MNIST dataset [LC98] with 8 million records, and is sampled such that on average each node will process 2-4GB of data. The data set of GSA is a sample of HCC1954, which is a single person’s whole genome. The input data for COMP is the first 100 kilo short reads in HCC1954.

The FPGA accelerators for all applications are designed in-house. The accelerator specifications for LR and KM can be found in [CHW16], and the Smith-Waterman implementation is based on [CCL15a]. Our FPGA accelerator is designed based on the Gzip implementation with both the LZ77 algorithm and Huffman encoding. Table 4.5 presents an overview of the accelerator speedup compared to the 12-thread CPU software baseline in terms of throughput improvement. We set \texttt{--num-executors} to 1 and \texttt{--executor-cores} to 12 in Spark.

**Table 4.5: FPGA accelerator performance profile**

<table>
<thead>
<tr>
<th>Application</th>
<th>Kernel</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>Gradients</td>
<td>3.4×</td>
</tr>
<tr>
<td>KM</td>
<td>DistancesSum</td>
<td>4.3×</td>
</tr>
<tr>
<td>GSA</td>
<td>SmithWaterman</td>
<td>10×</td>
</tr>
<tr>
<td>COMP</td>
<td>Deflater</td>
<td>26.7× over Gzip</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3× over Snappy</td>
</tr>
</tbody>
</table>
Table 4.6: Comparison of accelerator deployment efforts in terms of lines-of-code (LOC) changes

<table>
<thead>
<tr>
<th></th>
<th>App</th>
<th>ACC Setup</th>
<th>Partial FaaS*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LR</td>
<td>26</td>
<td>104</td>
<td>325</td>
</tr>
<tr>
<td>KM</td>
<td>37</td>
<td>107</td>
<td>364</td>
</tr>
<tr>
<td>GSA</td>
<td>0†</td>
<td>227</td>
<td>896</td>
</tr>
<tr>
<td>COMP</td>
<td>0†</td>
<td>70</td>
<td>360</td>
</tr>
<tr>
<td>Blaze</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LR</td>
<td>9</td>
<td>99</td>
<td>0</td>
</tr>
<tr>
<td>KM</td>
<td>7</td>
<td>103</td>
<td>0</td>
</tr>
<tr>
<td>GSA</td>
<td>0†</td>
<td>142</td>
<td>0</td>
</tr>
<tr>
<td>COMP</td>
<td>0†</td>
<td>65</td>
<td>0</td>
</tr>
</tbody>
</table>

*Partial FaaS does not support accelerator sharing among different applications, compared to full FaaS.
†In both GSA and COMP, the accelerator kernels are wrapped as a function replacing the original software version, so no software code change is counted.

For COMP, 12 independent streams on the CPU are used to take advantage of all cores. The accelerator design details are omitted in this thesis, since our focus is on evaluating the integration benefits of FPGA accelerators into big-data applications using Blaze.

Currently, we only run the kernel computation on FPGAs for FPGA-related experiments as a proof-of-concept. We will consider efficient CPU-FPGA co-working in our future work, which will provide higher performance than our current reported results.

### 4.2.4.2 Programming Efforts

We begin the analysis by showing Blaze’s benefits in reducing the deployment efforts of integrating existing FPGA accelerators to big-data applications. The results are shown in Table 4.6, where the lines of code (LOC) breakdown is listed for the selected applications.
The hardware code to design the accelerators is exactly the same between manual and Blaze implementations and decoupled from software developers, so it is excluded in this comparison. As an illustration of complexity of accelerator designs, it usually takes an experienced hardware engineer around 4 to 24 weeks to implement an efficient FPGA accelerator kernel, which is a big overhead for big-data application developers like Spark programmers. The design of LR, KM, GSA, and COMP accelerators take a senior graduate student 4, 4, 24, and 16 weeks to implement and optimize. Column ‘App’ in Table 4.6 shows code changes needed to modify the big-data applications so as to access accelerators in the application code. Column ‘ACC-setup’ shows the code changes for PCIe data transfer and accelerator invocation through OpenCL. Finally, column ‘Partial FaaS’ shows the code changes needed to enable sharing accelerators among multiple threads within the application.

Although using LOC to represent the programming efforts is not entirely accurate, it provides a rough illustration of the difference between each implementation method. Among the breakdown of LOCs, most of the “ACC-setup” code for accelerator control can be reused as long as the accelerator is fixed. We can see that deploying FPGA accelerators in big-data applications using Blaze is very easy, with less than 10 LOC changes in the application, and a one-time 100 LOC changes for accelerator setup. Without Blaze, even a manual design for partial FaaS to support accelerator sharing among multi-threads within a single application requires 325 to 896 LOC changes for every application.

4.2.4.3 Overall System Performance and Energy Gains for Single Application

Figure 4.6 demonstrates the single-node system speedup and energy reduction for our application case studies using Blaze and FPGA accelerators. For each individual job, we measure the overall job time and estimate the overall energy consumption based on the average power measured during application runtime. As mentioned earlier, we only run the kernel computation on FPGAs. Compared with the CPU baseline, the system with FPGA achieved $1.7 \times$

---

3For Figure 4.6, 4.7, and 4.8, the experiments are done by configuring `--executor-cores` to 12 and `--num-executors` to the number of nodes in Spark.
Figure 4.6: Single-node system performance and energy gains for each individual application.

to 3× speedup on overall system throughput, and 1.5× to 2.7× improvement on system energy consumption. (Note that FPGAs introduce an additional 25 watts per node into the system; therefore the achieved energy efficiency is slightly smaller than the performance speedup numbers.) This confirms that computation-intensive big-data applications can take full advantage of FPGA acceleration with Blaze.

Moreover, we compare the performance of a 4-node cluster with FPGAs to the CPU-only clusters with 4-node, 8-node, and 12-node. As shown in Figure 4.7, for LR and KM, a 4-node cluster with FPGA accelerators enabled can provide roughly the same throughput as a cluster of 12 CPU nodes. This indicates that we can reduce the conventional datacenter size by 3× by putting an FPGA into each server node, while achieving the same throughput.

Finally, Figure 4.8 presents the execution time breakdown of Spark jobs (the entire application instead of the kernel task execution time) on a 4-node cluster before and after FPGA acceleration. The results confirm that machine learning workloads such as LR and KM are computationally intensive, and the computation kernels benefit from FPGA acceleration. Note that the data load and preprocessing part in the original Spark program remain on the CPU, i.e., it is not accelerated by FPGA.
Figure 4.7: Performance of LR and KM on multiple nodes. The X-axis represents the experiment configurations. For example, "4N×12T CPU" represents the configuration of 4 CPU-only nodes with 12 threads on each node.

4.2.4.4 FaaS Overhead Analysis

To evaluate the potential overhead that Blaze introduces to provide FaaS, we evaluate the performance of Blaze integration against a reference manual integration. To make the analysis simple, we focus on the streaming COMP application. We first measure the normalized compression throughput to the reference manual design for 1-core and 12-core cases. As shown in Figure 4.9(a), for the two software baselines, the native Snappy implementation is around 10× faster than the Java Gzip implementation. For the single-core version, a manual integration of the compression FPGA accelerator achieves around 8.5× speedup over Snappy, while a Blaze integration achieves around 5.6× speedup. When there are 12 cores, the fully parallelized software implementation gets significant speedup, while Blaze integration and manual integration achieve similar performance, which is 1.7× better than Snappy.

Then we analyze why Blaze integration has more overhead than manual integration in the single-core case. We break down the execution time into FPGA kernel execution, JVM-to-native and native-to-FPGA data movement, and private-to-shared memory movement in Blaze native. The detailed breakdown is illustrated in Figure 4.9(b). As we can see, Blaze introduces the overhead of moving data from application private memory to the Blaze shared memory, which is required to manage accelerator sharing by multiple applications and costs
Figure 4.8: Execution time breakdown for LR and KM before and after FPGA acceleration on multiple nodes.

around 50% more execution time. Figure 4.9(b) also confirms that the overwhelming JVM-to-FPGA communication overhead occupies 76% of the total execution time in the single-core COMP application. Due to the multi-thread nature in big-data applications, such overhead can be alleviated using task pipelining (and data caching) that is transparently supported by our FaaS framework in Blaze. As a result, we see a comparable performance between Blaze integration and manual integration when there are 12 cores.

### 4.2.4.5 Breakdown of FaaS Optimizations

We show the breakdown of performance improvements by each JVM-to-FPGA communication optimization in Figure 4.10. We start from a naive FaaS without task pipelining or data caching, and then gradually add task pipelining and data caching. For each FaaS setup, we evaluate the FPGA kernel time and the task time. The task time represents the targeted accelerating kernel instead of the entire application, which includes both the time of data transfer to and from FPGA via PCIe and FPGA kernel time. FPGA kernel time stays the same across different cases since the total computation that needs to be performed on the FPGA remains the same.

As shown in Figure 4.10, a naive offloading of workload to accelerator may result in a
Figure 4.9: Faas overhead analysis in COMP application.

Figure 4.10: Breakdown of the JVM-to-FPGA communication optimizations in Faas.

slow-down rather than a speedup, e.g., 6.71× slowdown for LR and 6.95× slowdown for KM, due to the aforementioned JVM-to-FPGA overhead. By enabling data pipelining, the total time can be accelerated by a factor of 2.8× to 3.8×. For iterative computation of LR and KM, data caching provides a huge performance improvement since most of the data transfer is mitigated. Since all the data in GSA and COMP is processed only once, the results with and without data caching are identical, and thus omitted in Figure 4.10.

The benefits of task pipelining and data caching can be better illustrated using the accelerator utilization metric. In Figure 4.11 we show the different utilization patterns of running a single application LR on an FPGA. The accelerator utilization is defined as the ratio of accelerator execution time in a sampled interval of application execution time. The accelerator utilization is consistently low in the case without caching or pipelining shown in
the first part of the figure, since the accelerator keeps waiting for data to be transferred from the application. In the second part, when pipelining is enabled, the accelerator can reach high utilization periodically. This is because at the beginning of each iteration the first batch of data needs to be transferred before the accelerator can start, but once the pipeline begins, the accelerator can be kept busy with data continuously flowing in. Once data caching is enabled, the accelerator utilization can be increased dramatically. Similar results can also be observed for KM workloads as well. The high accelerator utilization in full-featured FaaS for KM and LR applications confirms again that the Blaze runtime overhead is negligible.

### 4.3 Conclusions

In this chapter we first discussed the challenges and opportunities of enabling FPGA-based accelerators as one of the building blocks in future datacenters to break the energy wall of scaling. Several different kinds of accelerator-rich system configurations are explored with prototyping systems to evaluate each configuration with real-world applications. As future datacenters will very likely be composed of different combinations of big-core, small-core and accelerators, resource allocation on heterogeneous systems can be a challenging problem to investigate.
To address the challenges of the complexity of future heterogeneous datacenters, we present the design and implementation of Blaze, which provides programming and runtime support that enables rapid and efficient deployment of FPGA accelerators at warehouse-scale. Blaze abstracts FPGA accelerators as a service (FaaS), decouples the FPGA accelerator development and big-data application development, and provides a set of clean programming APIs for big-data applications to easily access the performance and energy gains of FPGA accelerators. In the FaaS framework, we provide efficient accelerator sharing by multiple heterogeneous threads, hide the overwhelming Java-to-FPGA data communication overhead, and support fault tolerance. We implement FaaS as a third-party package that works with Apache Spark. In addition, we propose to manage the logical accelerator functionality as a resource instead of the physical hardware platform itself. Using this new concept, we are able to extend Hadoop YARN with an accelerator-centric scheduling policy that better manages global accelerator resources and mitigates the FPGA reprogramming overhead.

Our experiments with four representative big-data applications demonstrate that Blaze greatly reduces the programming efforts, and improves the system throughput from $1.7\times$ to $3\times$, i.e., a $1.7\times$ to $3\times$ datacenter size reduction using FPGAs with the same throughput. We also demonstrate that our FaaS implementation achieves performance similar to a manual design under the dominant multi-thread scenarios in big-data applications, while our accelerator-centric scheduling achieves close to optimal system throughput.
CHAPTER 5

Efficient Job Scheduling in Large-Scale Accelerator Runtime Systems

Chapter 4 discusses how to deploy accelerators at datacenter-scale for scale-out applications. One issue that is not properly addressed is how to properly distribute the workloads between the general-purpose CPUs and FPGA accelerators. In this chapter, the Blaze runtime system in Chapter 4 is extended using the scale-up design methodology mentioned in Chapter 3 to maximize the system utilization and application performance.

5.1 Introduction

In an FPGA-enabled cloud platform, such as Amazon Web Services [ama], users typically pay for their on-demand execution time per computing instance that usually includes a multicore CPU and one or more FPGAs. For example, an Amazon EC2 F1.2xlarge instance includes an 8-core CPU and a PCIe-based Xilinx UltraScale+ VU9P FPGA [ama]. As a result, it is very important to fully utilize all CPU cores and FPGA resources within such a requested instance, so as to improve the ratio of performance per cost.

Unfortunately, many prior studies in FPGA acceleration of big-data and streaming applications [WLW17, CHL12, CHZ14, HKM08, FKB15, CCL15b] mainly focus on the accelerator design itself and do not pay enough attention to the accelerator integration for end-to-end system performance. Blaze and other frameworks such as [ABB12] provide programming and runtime support to ease the FPGA accelerator integration into big-data applications and optimize the CPU-FPGA communication overhead. However, they usually leave the CPU cores idle while offloading computation to the FPGA accelerators. Although
some studies [CCC16, CFH17, ASH15] notice this problem and perform CPU-FPGA co-optimization for their applications, such approaches are limited to one specific application and require a notable amount of manual efforts for the efficient CPU-FPGA integration and co-optimization.

Multiprocessor and heterogeneous platform task scheduling is a well-studied topic. Many frameworks such as FastFlow [ADK14] and runtime systems such as [Kes12, Kim11, Lee13, Luk09] have been proposed to improve both programmability and efficiency of parallel task scheduling. However, most existing studies are focused on scheduling for the computing kernels—not the entire program—and many focus on dispatching jobs to a fixed set of processors. In the study of [SZB12], a framework is proposed to dynamically schedule CnC (Concurrent Collections) programs onto multicore CPUs, GPUs, and FPGAs using dynamic work stealing. However, it requires users to rewrite the entire program in a new language.

Here we propose an extension to the Blaze runtime system, which is called K-Flow. K-Flow is designed with the goal to automatically optimize the CPU-FPGA resource utilization and thus the overall system performance per cost for big-data applications running in the FPGA-enabled cloud.

1. K-Flow is based on a widely used dataflow model that can be expressed as a directed acyclic graph (DAG). This means it is compatible with most existing big-data frameworks such as Apache Spark [ZCF10] and TensorFlow [Aba15]. With the dataflow abstraction, K-Flow provides a clean separation between CPU and FPGA execution, and automates the co-optimization between CPU and FPGA execution.

2. Unlike many existing systems that schedule tasks to the processors, the K-Flow runtime system dynamically schedules available CPU processors and FPGAs to tasks based on the program DAG. With a simple scheduling scheme based on list scheduling of the DAG stages, K-Flow can achieve the theoretical upper bound of Amdahl’s Law [Amd67] for a dataflow program with marginal accelerator speedup. Moreover, K-Flow can guarantee a speedup after FPGA integration—even with a slow accelerator design.
3. K-Flow also provides a set of user-friendly APIs in C++ to program the DAG representation, which enables efficient program transformation for acceleration.

To demonstrate the effectiveness of K-Flow, we conduct a case study to accelerate the widely used DNA read alignment program BWA-MEM [Li13b] on a CPU-FPGA platform using K-Flow. Experimental results show that it only takes 797 lines of code to implement BWA-Flow (the K-Flow version of BWA-MEM) on a CPU-FPGA platform with a pre-designed FPGA accelerator, which is negligible compared to the 16K+ lines of code in the original BWA-MEM software. On average, K-Flow achieves 97.6% of the theoretical optimal throughput, and 1.6× speedup compared to the straightforward FPGA integration.

5.2 Motivation

Big-data applications usually have a large number of data partitions that can be processed in parallel. For example, in a MapReduce [ZCF10] program, data partitions are processed concurrently as map tasks on different CPU threads. Consider a program shown in Figure 5.1, which consists of only map tasks. We can model its performance using throughput $T = \frac{P}{r}$. 

![Figure 5.1: Baseline CPU program with multiple iterations for all $N$ data partitions: The shadow segments indicate the part of the program that can be accelerated. In this case, both the white and shadow segments run on CPU. Throughput $T = \frac{P}{r}$.

Big-data applications usually have a large number of data partitions that can be processed in parallel. For example, in a MapReduce [ZCF10] program, data partitions are processed concurrently as map tasks on different CPU threads. Consider a program shown in Figure 5.1, which consists of only map tasks. We can model its performance using throughput $T$, which represents the number of data partitions processed within a unit of time. For simplicity, we assume each data partition has the same size, and each map task has the same
Figure 5.2: Straightforward integration with fast FPGA: White segments on CPU and shadow segments on FPGA. The runtime of all $P - 1$ shadow segments is not longer than that of 1 white segment. FPGA speedup $S = \frac{r(P-1)}{1-r}$. Throughput $T = \frac{P}{S + (1-r) \cdot t}$. Problem: CPU is idle during FPGA execution for shadow segments.

execution time. In the example program, assume $t$ is the run time of a sequential map task for one data partition; it can process $P$ partitions in $t$, where $P$ is the number of parallel threads in the CPU. Assume the total number of data partitions is $N$ (typically much larger than $P$), the total runtime to finish $N$ partitions is $N \cdot t$. Therefore, the throughput of the example program can be defined as: $T = \frac{N}{T} = \frac{P}{T}$, where $T$ is independent of $N$.

A portion $r$ of each map task can be accelerated with an FPGA accelerator, which is illustrated as the shadow segments in Figure 5.1. In a straightforward FPGA integration such as Blaze, each CPU core will offload its $r$ portion of the task onto the FPGA. After the task (shadow segment) is offloaded, the CPU core then polls for the FPGA accelerator to finish before processing its next data partition with $1 - r$ portion (white segment). An example flow of the straightforward integration with a fast FPGA accelerator is shown in Figure 5.2. Here we assume there is only one big FPGA accelerator and each core sequentially offloads a task to the FPGA acceleration in a pipeline fashion. We use $S$ to denote the speedup of the FPGA accelerator against a single core.\(^1\) We can write down the throughput $T$ of the

\[^1\text{In our formulation we only care about the end-to-end FPGA accelerator speedup } S, \text{ which means } S \text{ already includes the CPU-to-FPGA data transfer overhead.}\]
Figure 5.3: Straightforward integration with slow FPGA: White segments on CPU and shadow segments on FPGA. The runtime of all $P−1$ shadow segments is longer than that of 1 white segment. In this case FPGA becomes the bottleneck when the speedup $S ≥ \frac{r(P−1)}{1−r}$. Throughput $T = \frac{S}{r\cdot t}$. Problem: CPU is idle during FPGA execution and FPGA becomes the bottleneck. It can be even worse than the case in 5.1 when $S ≥ r\cdot P$.

Two different scenarios are considered in Equation 5.1.

1. When the FPGA accelerator is fast enough, such that the runtime of all $P−1$ shadow segments on FPGA is no longer than that of 1 white segment on CPU, as shown in Figure 5.2. That is $(P−1)\cdot \frac{r\cdot t}{S} ≤ (1−r)\cdot t$, or $S ≥ \frac{r(P−1)}{1−r}$. In such a case, Equation 5.1 is equivalent to Amdahl’s Law [Amd67], and the overall throughput improvement is limited by $r$. The inefficiency here is that when the FPGA is executing the tasks (shadow segments), the CPU threads are just idle waiting for the FPGA to finish.

2. When the FPGA accelerator is slow, i.e., $S < \frac{r(P−1)}{1−r}$, the FPGA becomes the bottleneck of the entire program. As shown in Figure 5.3, the offloaded task of core 0 in the current iteration needs to wait for the offloaded task of core $P−1$ in the previous iteration to finish. Thus, the throughput is simply the overall FPGA throughput $\frac{S}{r\cdot t}$. 
Figure 5.4: Theoretically optimal integration with fast FPGA: White segments on CPU and shadow segments on FPGA. This corresponds to the case in 5.2, and we omit the optimal case for 5.3. Throughput $T = \frac{P}{(1-r)t}$.

In this situation, the CPU wastes more time idling during the FPGA gap. In fact, it may even achieve a lower throughput than the baseline CPU program, when $\frac{S}{r_t} < \frac{P}{t}$, i.e., $S < r \cdot P$. This, in reality, significantly limits the adoption of FPGA accelerators in big-data applications.

In contrast to the straightforward integration, the optimal scheduling should always allow the CPU cores to work concurrently with the FPGA, and when the FPGA is not fast enough some CPU cores can be used to execute those tasks. A theoretically optimal integration flow for the fast FPGA case is illustrated in Figure 5.4. In this case the throughput $T = \frac{P}{(1-r)t}$, which is also the upper bound of Amdahl’s Law. With the slow FPGA case, some of the shadow tasks need to be executed on CPU to achieve the maximum throughput. We omit the illustration for this case.

Many related works in the literature use techniques such as asynchronous FPGA task invocation to approximate the optimal scheduling. However, these techniques are often application-specific, and require a considerable amount of manual expertise to perfect. Moreover, in real cases the FPGA speedup $S$ is usually data-dependent and difficult to predict, which makes it very difficult to statically partition the workloads between CPU cores and FPGA.
To address these challenges, K-Flow is created to automate the FPGA accelerator integra-
tion in a dataflow fashion and optimize the CPU-FPGA co-execution through dynamic
job scheduling. Next, we show that K-Flow can achieve the schedule shown in Figure 5.4.

5.3 K-Flow: Dynamic Job Scheduling for CPU-FPGA Co-Execution

The key innovation of K-Flow is the dynamic scheduling of the dataflow execution on mod-
ern CPU-FPGA platforms that achieve the optimal system throughput. We first introduce
the basic primitives of the K-Flow programming model based on the widely used representa-
tion of directed acyclic graph (DAG), which is commonly used by many existing big-data
frameworks such as Apache Spark [ZCF10] and TensorFlow [Aba15].\(^2\) We argue that the
scheduling part of K-Flow is versatile to any existing DAG-based big-data frameworks. To
evaluate the scheduling system of K-Flow, we provide a set of user-friendly APIs in C++
to help programmers describe the DAG and automate the process of FPGA integration and
scheduling.

5.3.1 Programming Model Overview

K-Flow uses a dataflow programing model that is different than traditional program execu-
tion flow to efficiently integrate FPGA accelerators into an existing program. To explain
this, we use an example dataflow program shown in Figure 5.5a. In this program, multi-
ple data partitions are processed in parallel with the same computation. The shadow area
marks the region in the program that can be accelerated with an FPGA. To use K-Flow to
integrate an FPGA accelerator, this program is separated into three different stages, with
the second stage being the acceleratable region. The new program can be abstracted as a
DAG shown in Figure 5.5b. In the DAG, each node represents a computing \textit{stage}, and each
directed edge denotes the direction that the data flows through between stages. Using the
same denotation, the original program in Figure 5.5a is also a DAG with a single node.

\(^2\)We can model the cycles in a graph as a single node to make it a DAG.
Acceleration of the shadowed region is done by adding into the original DAG a new branch that represents a separate path for the data to flow through the added accelerators. The new DAG with FPGA acceleration is shown in Figure 5.5c, where stage 3 is added for FPGA execution. In the new DAG, output data partitions of stage 1 can be processed by both the CPU stage and the FPGA stage.

Instead of using all CPU cores to execute data partitions for one DAG node (i.e., computation stage) at a time, K-Flow dynamically assigns part of the CPU and FPGA threads to each DAG node and executes all nodes concurrently in a pipelined fashion. In this way, FPGA execution and CPU execution will be overlapped similarly, as in Figure 5.4.

In summary, here is a list of advantages provided by the K-Flow programming model:

1. The K-Flow DAG is equivalent to the original program, with all the sequential dependencies preserved. So no algorithm change is required in a K-Flow implementation.
2. With this model, the scheduler can coordinate the parallel execution of the program
without violating the semantics of the original problem.\(^3\)

Once we have such a K-Flow DAG representation for a program, either from existing frameworks such as Spark or from the K-Flow APIs, K-Flow can automatically schedule the available CPU and FPGA resources for the optimal dataflow execution, as shown in the next section.

### 5.3.2 Dynamic Job Scheduling

K-Flow automatically interprets the DAG representation introduced in Section 5.3.1 and realizes the implementation. In K-Flow, it implements each node in a DAG as a function, which defines (by users) the computation that consumes one or multiple data partitions. Each invocation of a function is called a task, which is executed by a logical thread that is either running on a CPU core or an FPGA processing element (PE). The directed edges in the DAG are implemented as lock-free queues, in which different stages can push and pull data partitions asynchronously. The DAG nodes that share the same source (or destination) node will share the same input (or output) queues. An example K-Flow implementation of the example program in Figure 5.5c is illustrated in Figure 5.6.

An overview of the K-Flow automatic task scheduling framework will be described in Section 5.3.2. The key design strategy is to keep all the available hardware resources in the system fully utilized and avoid blocking task execution. Before presenting the dynamic scheduling algorithm, we first explain the basic concepts of tasks and thread pools in K-Flow.

**Tasks** K-Flow provides two primitives of task functions based on the way it consumes data. The first primitive is called map function, which is shown in Figure 5.7a. The map function takes exactly one data partition (more exactly, one data partition of all its input data) as input, performs all computation for this single data partition, and produces one output data partition. The map task is widely used for parallel processing. For the example shown in

\(^3\)The model of K-Flow does not restrict strict ordering of the different input data partitions. In fact, K-Flow provides APIs that support data reordering if strict execution order is required.
Figure 5.6: Implementation of an example K-Flow program.

Figure 5.1, all the tasks are map tasks. The second primitive is called shuffle function, shown in Figure 5.7b, which takes zero or more data partitions as input and produces zero or more data partitions. The shuffle tasks are used in scenarios where the data partitions need to be reorganized, such as combining, splitting, and re-ordering. We also use shuffle tasks for I/O, in ways such that multiple data partitions can be extracted from a single file, or multiple data partitions are serialized and written to a file.

(a) Map function  (b) Shuffle function

Figure 5.7: Primitive function/task types in K-Flow.

**Thread Pools** K-Flow manages a separate thread pool for each device in the system. In the example shown in Figure 5.8, two thread pools are assigned: one for CPU and one for FPGA. The number of logical threads in the thread pool can be assigned by users, and by default, and is based on the available hardware resources. For CPUs, the thread number
is typically the total number of cores. For FPGAs, it is the number of parallel processing elements (PE) that are synthesized. Usually when FPGA devices are integrated, one thread will be deducted from the CPU thread pool to accommodate the FPGA runtime system. In the K-Flow implementation, a programmer can indicate the type of pool a stage requires. In the example of Figure 5.8, stage 3 is executed by the FPGA pool while all other stages are executed by the CPU pool.

![Figure 5.8: Overview of K-Flow dynamic scheduling system.](image)

**Task Scheduling** To implement the dynamic task scheduling, K-Flow first constructs a *scheduling queue* by topologically sorting all the nodes in the program DAG, with the final stage/node being the back of the queue and the first stage being the front. This is illustrated in Figure 5.8 with the same stage numbering in Figure 5.6.

A high-level description of the scheduler is shown in Algorithm 1. During runtime, the K-Flow task scheduler iteratively scans this scheduling queue to assign CPU and FPGA threads for the task execution. It starts assigning threads to the stage at the back of the queue, and stops the iterative scheduling until there is no input data partition for the stage at the front of the queue. In other words, the later stages will be executed first as long as there are remaining input data. This scheme guarantees deadlock-free execution such that
the earlier stages will never be blocked by later stages.

When there are two stages sharing the same input data queue, the ordering of them in the scheduling queue is determined by a user-assigned priority. The stage with higher priority will be pushed to the scheduling queue after the stage with lower priority. By default, this priority is determined by the speedup of the device executing the stage: a stage with a faster device has higher priority. In the example in Figure 5.8, the FPGA stage 3 is placed after the CPU stage 2, which means tasks will be preferred for execution by the FPGA unless it is busy. This design helps maximize the utilization of faster devices, which in turn maximizes the system throughput.

In the beginning of each scheduling iteration, the scheduler will wait until all the available threads are scheduled. Once a thread finishes its task it will send an asynchronous interrupt to the scheduler, which can then proceed. Once there is an available thread, the scheduler makes the thread assignment decisions for the current stage based on its input data queue. The key strategy is to make sure none of the assigned threads from the pools are idle waiting for input data.

1. If the current stage is a map task and its input data queue is not empty, one thread from the corresponding thread pool will be assigned to this task. Upon assignment, one data partition will be consumed from the input data queue, and then the scheduler moves on to the next stage in the scheduling queue. Meanwhile, with the input data partition ready, the assigned thread will start computation immediately. After the task is finished, the thread will be returned to the thread pool, and can be recycled for other stages. In this scheme, the available threads, i.e., CPU cores and FPGA resources, will be automatically distributed across all the stages.

2. If the stage is a shuffle task, the thread assignment will be slightly different because the number of data partitions consumed by a shuffle task may be variable. Therefore, K-Flow can not determine if the shuffle task will be blocked waiting for input data partitions during runtime. The solution is to use logical thread tokens to indicate whether a task is executing or being blocked. When K-Flow assigns a thread to a
while scheduling.queue is not empty do
    if all thread pools are empty then
        wait for an idle thread to be available;
    end
    scheduled ← false;
    for stage ← scheduling.queue.back to scheduling.queue.front do
        if stage has no input queue or input queue is not empty then
            switch stage.type do
                case FPGA do
                    if FPGA pool is not empty then
                        assign thread from FPGA pool;
                        scheduled ← true;
                    end
                end
                case CPU do
                    if CPU pool is not empty then
                        assign thread from CPU pool;
                        scheduled ← true;
                    end
                end
            end
        end
        if scheduling.queue.front is finished then
            scheduling.queue.pop_front;
        end
        if scheduled is false then
            sleep for ts;
        end
    end
end

Algorithm 1: An overview of K-Flow task scheduling algorithm.
shuffle stage, besides allocating a physical thread it also sends a token. When the task is blocked trying to pull data from an empty input data queue, it sends the token back to the K-Flow scheduler and goes to hibernation. With the token back, the K-Flow scheduler is free to schedule this additional thread to other stages. The shuffle stage gets the token back from the scheduler at the next iteration, and it will then wake up and resume the execution again.

At the end of each scheduling iteration, the scheduler checks whether the front stage of the scheduling queue is finished. If so, the stage will be popped out of the queue. A stage is considered finished if and only if: 1) there is no active thread in that stage, 2) the input queue is empty if the stage has an input queue, or there is no more input data from IO if the stage has no input queue (i.e., it directly reads from IO), and 3) all the preceding stages are finished. In this way, a finishing signal will be propagated from the start to the end of the pipeline. Once all the stages are finished, the entire program finishes. In addition, the scheduler also checks whether any thread is scheduled at all in the iteration. If not, the scheduler will sleep for a short period of time ts to avoid constantly spinning.

In the K-Flow runtime, the task scheduler is a lightweight thread that wakes up periodically whenever there is an idle thread in the pools. By recording the next non-empty stage in the scheduling queue, each job can be scheduled in constant time. Therefore, the scheduling overhead is insignificant—especially since the most big-data applications have tasks that run on large data partitions with long execution time.

5.3.3 Programming Interface

To evaluate the scheduling system and to support an easy FPGA integration, K-Flow also provides a user-friendly programming interface for big-data applications that can be expressed in a DAG-based dataflow model. From a programmer’s perspective, only the sequential computation for each DAG node and the shape of the DAG (i.e., directed edges between DAG nodes) need to be defined. The K-Flow runtime will automatically interpret the DAG and takes care of the dynamic task scheduling as explained in Section 5.3.1
The current K-Flow programming interface is based on C++. The function primitives in K-Flow are created as base classes for users to overwrite. Here we explain the primitives for *map* and *shuffle* functions defined in Section 5.3.1 using a simple vector add example in C++ shown below:

```c++
// load vector_a and vector_b as float[]
for (int i = 0; i < size; i++) {
    vector_c[i] = vector_a[i] + vector_b[i]
}
// write vector_c
```

**Map function.** For our vector add example, the *map* task computes the element-wise addition. To define the functionality, a programmer simply needs to overwrite the `compute` function of the base class, which takes one input data partition of type \( T \) and produces an output data partition of type \( U \). The detailed definition of the data types (e.g., \( T \) is a pair of arrays, \( U \) is an array) of this example is omitted here.

```c++
class Add : public Map<T, U> {
    U compute(T const & in) {
        // perform addition of two elements
    }
};
```

**Shuffle function.** Similar to the *map* function, the *shuffle* function is also expressed by extending a provided base class. The example below is based on vector add. The `get_input()` and `put_output()` are K-Flow APIs to get input data partitions and push output data partitions, respectively. With *shuffle* functions, the programmer has the freedom to express different data access patterns. These function calls will be blocking if the executing token is sent back to the scheduler.

```c++
class Collect : public Shuffle<U, V> {
    void compute() {
        U in; V out;
        while (get_input(in)) {
```
Pipeline. After defining all the DAG nodes, the programmer needs to add them to a pipeline class that specifies the directed edges of all DAG nodes. Then the API pipeline.run() is invoked to start the computation. The entire program flow is driven by data, which means it will terminate once all the data is finished processing. A finishing signal will be propagated from the first stage to the final stage.

FPGA integration. The programming model of K-Flow makes it easier to integrate multiple heterogeneous platforms, such as FPGAs, GPUs, and ASICs. Adding another platform is as simple as overwriting another function primitive (DAG node) and hooking it up with the existing pipeline. A programmer does not need to figure out how to schedule tasks between different stages or different platforms. K-Flow will automatically balance the FPGA and CPU threads among all stages. When defining an FPGA computation stage, we further leverage Blaze which abstracts FPGA accelerators as a service in order to reduce the FPGA integration efforts.

5.4 Theoretical Analysis

5.4.1 Throughput Upper Bound

To analyze the scheduling scheme of K-Flow, we first formulate the throughput upper bound using a DAG. Using the same example in Figure 5.5b, we let \( r_i \) and \( P_i \) denote the ratio of execution time and number of CPU threads allocated for stage \( i \), with \( i = \{1, 2, 3\} \). Then the throughput \( T_i \) for a CPU stage \( i \) is \( \frac{P_i}{r_i t} \). Suppose the FPGA speedup compared to a single CPU thread is \( S \), then we can summarize the throughputs for each stage in Figure 5.5c as
follows:
\[
T_1 = \frac{P_1}{r_1 \cdot t}, \quad T_3 = \frac{S}{r_2 \cdot t} \\
T_2 = \frac{P_2}{r_2 \cdot t}, \quad T_4 = \frac{P_3}{r_3 \cdot t}
\] (5.2)

Based on the definition of the dataflow model used by K-Flow, we can derive the overall system throughput for Figure 5.5c as \( \min(T_1, T_2 + T_3, T_4) \). Then the following linear programming problem can be used to maximize the overall system throughput given \( r_1, r_2, r_3 \) and \( S \):

\[
\begin{align*}
\text{maximize} & \quad T \\
\text{subject to} & \quad T \leq \frac{P_1}{r_1 \cdot t} \\
& \quad T \leq \frac{P_2}{r_2 \cdot t} + \frac{S}{r_2 \cdot t} \\
& \quad T \leq \frac{P_3}{r_3 \cdot t} \\
& \quad P_1 + P_2 + P_3 \leq P \\
& \quad r_1 + r_2 + r_3 = 1
\end{align*}
\] (5.3)

The formulation can easily be extended to arbitrary DAGs by modeling the throughput of each node, and lay out the inequalities based on the connectivity of the DAG nodes.

To be consistent with the discussion in Section 5.2, we use \( r \) which represents the FPGA accelerated ratio in the solution, with \( r = r_2 \). The solution for Equation 5.3 satisfies \( T = T_1 = T_2 + T_3 = T_4 \), and maximum throughput \( T \) can be calculated as:

\[
T = \begin{cases} 
\frac{P + S}{t} & \text{if } S \leq \frac{P \cdot r}{1 - r} \\
\frac{P}{(1 - r) \cdot t} & \text{if } S > \frac{P \cdot r}{1 - r}
\end{cases}
\] (5.4)

5.4.2 Optimality of K-Flow Scheduling

The scheduling problem for systems like K-Flow can be modeled as the multiprocessor task scheduling problem, which is NP-hard \([GJ90]\). Therefore, the upper bound throughput cannot be guaranteed given arbitrary tasks. However, we can argue that K-Flow achieves maximum CPU and FPGA utilization for streaming applications with endless incoming data, which in turn demonstrates the efficiency of the scheme.
When $S$ is larger than $\frac{P \cdot r}{1 - r}$—in other words, it is faster than the combined throughput of all the CPU threads—the optimal throughput is $P \cdot \frac{1}{(1 - r) \cdot t}$. It can be easily calculated that the throughput can be achieved when all CPU threads are fully utilized working on stage 1 and 4. K-Flow achieves this threads assignment based on its priority-based scheduling. Suppose a CPU thread is scheduled to stage 2, then the combined throughput of stage 2 and stage 3 will be larger than the optimal throughput. Therefore, either the input queue for stage 4 starts filling up, or the input queue for stage 2 and 3 becomes empty. In either cases, idle threads will be allocated to stage 4 or stage 1 in the next scheduling iteration. When equilibrium is reached, the thread assignment will become optimal.

When $S$ is smaller than $\frac{P \cdot r}{1 - r}$, the optimal throughput is $\frac{P + S}{t}$, which indicates that both the CPU and FPGA are fully utilized and can achieve their maximum throughput. We can use a similar equilibrium analysis to argue that K-Flow achieves the same thread assignment. In this case, FPGA will be fully utilized based on the analysis in the previous paragraph. CPU threads will also be fully utilized since an idle thread can always be scheduled to stage 1 if the input queues for each stage are empty.

It is apparent that the solutions $P_i$ for linear problems such as Equation 5.3 are fractional numbers, which means the threads cannot be statically partitioned to achieve the optimal throughput. K-Flow’s scheduler dynamically balances the thread occupancy for each stage, and when the number of data partitions $N$ is large enough it is equivalent to the optimal fractional partitioning of threads.

In the experiment section of this chapter, we evaluate the performance of K-Flow with a case study of a real-world application. Results show that the K-Flow scheduling algorithm can achieve the system throughput that is very close to the optimal solution of Equation 5.3.

### 5.4.3 Discussion on Throughput Upperbound

To better illustrate the improvement from our K-Flow model, we calculate the theoretical throughput improvement of both the straightforward and dataflow integrations of the FPGA, compared to the CPU baseline. The throughput improvement is plotted as a function of the
FPGA speedup $S$ with fixed $P$ and $r$ values. Figures 5.9a, 5.9b and 5.9c show the results for $r = 0.25$, $r = 0.5$ and $r = 0.75$, respectively. We find that the proposed dataflow integration always has a throughput improvement (larger than 1), and has a significant advantage over the straightforward integration when the FPGA speedup $S$ is low. Moreover, the K-Flow implementation can achieve the optimal throughput with small FPGA speedups, which makes it more practical to adopt FPGA accelerators.

Figure 5.9: Throughput speedups of straightforward integration and theoretical throughput of K-Flow with $P = 12$ and different acceleratable ratios $r$, compared to the CPU baseline. The X-axis is the FPGA accelerator speedup $S$ compared to a single CPU core.

5.5 Case Study: CPU-FPGA Acceleration for Genome Alignment

Next-generation DNA sequencing (NGS) technologies have made remarkable advances in the past decade, and the cost of sequencing an individual’s whole genome has dropped dramatically [ngs], i.e., dropping much faster than Moore’s law. With the sequencing cost going down to around $1,000, the computation of genome analysis, in terms of both computation performance and cost, is becoming the major limitation to apply NGS in clinic settings. In NGS, human DNAs are broken into small fragments which can be sequenced reliably. These
fragments are called short reads, and are typically 50 to 250 bases long. After the short reads are generated, they are then aligned to a reference genome to reconstruct the original sequence, and this reconstruction process is called genome read alignment, which is one of the most computation-intensive steps of genome data analysis.

One of the most widely used software packages for NGS read alignment is BWA-MEM [Li13b], which uses Burrows-Wheeler transformation to reduce the complexity of the alignment. Many prior studies [ASH15, CCC16, CCL15b, CCF16] use customized hardware such as FPGAs to accelerate different parts of BWA-MEM. Among them, the study in [ASH15] provides a full system acceleration of BWA-MEM with FPGA acceleration of the Smith-Waterman alignment (a dynamic programming algorithm for inexact alignment [Li13b]) and software optimizations for the other steps of the program. However, it assumes that the Smith-Waterman accelerator is always fast enough to completely overlap with CPU execution. We find in our experiments that this assumption is not always true, considering a much faster CPU and/or different input genome data sizes.

Here we conduct a case study of K-Flow of accelerate BWA-MEM on a CPU-FPGA platform, with the goal to achieving close-to-optimal system throughput. We will demonstrate that with K-Flow, integrating FPGA accelerators into BWA-MEM can be both simple and efficient.

BWA-MEM is a data streaming algorithm, where short reads are aligned concurrently in independent batches. The sequential algorithm of a single batch is shown in Figure 5.10a, with the computation partitioned into three different stages: seq2chain, chain2aln, and aln2sam. We first implement BWA-Flow, a K-Flow implementation of BWA-MEM on CPU. In BWA-Flow, the program DAG is the same as the sequential algorithm for a single batch, and each data item is exactly a single batch of short reads. The data read and write stages are implemented using the shuffle API with a single thread, and the three computing stages are implemented using the map API. In the CPU implementation of BWA-Flow, not a single line of the original BWA-MEM code was modified. We simply write wrapper classes and

4Each base is a nitrogen base in DNA, which is represented as a single letter: A (adenine), C (cytosine), G (guanine), T (thymine).
data structures in C++ using K-Flow APIs for the functions of the original BWA-MEM, which only takes 323 lines of code.

![Diagram](a) DAG for BWA-MEM and BWA-Flow on CPU

![Diagram](b) DAG for BWA-Flow with FPGA

Figure 5.10: DAGs for BWA-MEM and BWA-Flow

Next we integrate a Smith-Waterman FPGA accelerator based on [CCL15b] to accelerate the chain2aln stage in BWA-Flow. Figure 5.10b shows the DAG of our BWA-Flow implementation with FPGA. The Blaze runtime system is used to manage the FPGA access of the fpga-compute stage. An additional 474 lines of code is used to integrate the FPGA using Blaze, most of which is serializing (deserializing) the data into (from) FPGA format. With page limitations, the detailed implementation is omitted.

Overall, it takes only 797 (323 + 474) lines of code to implement BWA-Flow on a CPU-FPGA platform. Such additional change is negligible compared to the 16K+ lines of code in the original BWA-MEM software package. This case study demonstrates that K-Flow provides a very user-friendly interface for users to integrate pre-designed FPGA accelerators into their original software programs; this requires a very small amount of programming efforts. The overall system throughput will be evaluated next.
5.6 Experimental Results

In this section we first describe the experimental setup and then evaluate the overall system throughput improvement of K-Flow on both on-premises and cloud-based systems.

5.6.1 Experimental Setup

Our on-premises experimental platform is a high-end server with dual-socket Intel Xeon E5-2687Wv4 CPUs that have 24 physical cores in total and run at 3.0GHz and 256GB of main memory. The FPGA card in this system is a PCIe-based Xilinx KU115 card, which contains a Xilinx Virtex-7 XC7VX690T-2 FPGA chip and 16GB of on-board DDR3 memory. The FPGA board can be powered by PCIe alone and consumes around 45W of power. Because of the high-speed CPU available on this system, we can showcase the situation where an FPGA device is not significantly faster than a CPU.

Our cloud-based platform is based on the Amazon Web Service f1.2xlarge instance [ama]. Each of the f1.2xlarge instances contain 8 virtual CPU cores based on Intel Xeon E5-2686v4 running at 2.30GHz and 122GB of memory. The FPGA system on F1 is the Xilinx Ultra-scale+ VU9P FPGA, which has about 1.5× area compared to the KU115 FPGA card. The CPU cores on the F1 instance are much slower than our on-premises system, and the FPGA is faster. This provides us with another set of profiles to showcase K-Flow’s efficiency on different systems.

The BWA-Flow implementation is based on BWA-MEM version 0.7.13-r1126 [Li13b]. The FPGA implementation generated using Xilinx SDx 2017.1 and the system integration are done using the Blaze Runtime System.

The input datasets used in the experiments are high-coverage exome sequences and sampled whole genome sequences. All the data are selected from public domain including the 1000 Genome project [100] and GCAT (Genome Comparison and Analytical Testing) project [HWK15].
5.6.2 K-Flow Throughput on CPU

We first evaluate the efficiency of K-Flow by comparing the CPU-only implementations before and after applying K-Flow. The baseline implementation is the standard release of the BWA-MEM software package, and here the BWA-Flow is the CPU-only implementation using K-Flow. The details of BWA-Flow can be found in Section 5.5.

We compare the throughput of BWA-Flow against the baseline with different numbers of thread and different samples from our input datasets. The throughput improvement of BWA-Flow is shown in Figure 5.12. The case with a single thread is used to evaluate the overhead of K-Flow, and the results show a very minimum slowdown of 1.4%. Regardless of the number of threads, BWA-Flow achieves a very close performance to—sometimes even better than—that of the software baseline.

![Figure 5.11: Throughput improvement of CPU-only BWA-Flow compared to of the original BWA-MEM software, under different numbers of CPU threads (P).](image)

To better understand the efficiency of K-Flow, we also visualize the scheduling of threads in different stages of BWA-Flow. Since the middle three stages (seq2chain, chain2aln, aln2sam) of BWA-Flow dominates the entire execution time, we plot their execution of a sample experimental run in Figure 5.12, where the total number of threads \( P \) is set to 16. Since there is little variation of the execution times of each data partition, the over-
all program flow actually resembles the baseline BWA-MEM implementation. Note that in K-Flow, a latter stage (e.g., stage 2) can only start after the prior stage (e.g., stage 1) finishes processing one data partition and writes the output data partition into the data queue between these two stages.

Figure 5.12: Visualization of thread assignment/execution for one sample run of CPU-only BWA-Flow. The total number of CPU threads $P$ is set to 16. Each row in the figure represents the task assignment over time (seconds in X-axis) of an individual thread.

### 5.6.3 K-Flow Throughput on CPU-FPGA

As discussed in Section 5.5, the Smith-Waterman FPGA accelerator [CCL15b] is used to accelerate the chain2region stage of BWA-MEM. In practice, both the execution time ratio of this stage and the FPGA speedup depend on the input genome sample as well as the systems. In general, the FPGA accelerator can achieve higher speedup when the length of the short reads are longer. Table 5.1 shows the profile of the program for different input datasets on both the local server and the f1.2xlarge instance. The profile includes the execution time ratio $r$ of the chain2align in the overall program, as well as the FPGA speedup $S$ against a single-thread CPU implementation. The speedups on the F1 instance are larger than the local server because the FPGA kernel is faster, and also the CPU cores are slower than a local server. Also, as noted in previous sections, this speedup already has the data transfer overhead factored in. Therefore, the speedup $S$ is significantly smaller than
that reported in [CCL15b].

Table 5.1: Application Profile of BWA for different Datasets and Platforms

<table>
<thead>
<tr>
<th>Sample</th>
<th>local</th>
<th>f1.2xlarge</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ratio ($r$)</td>
<td>Speedup ($S$)</td>
</tr>
<tr>
<td>sample 1</td>
<td>0.195</td>
<td>3.63</td>
</tr>
<tr>
<td>sample 2</td>
<td>0.163</td>
<td>3.56</td>
</tr>
<tr>
<td>sample 3</td>
<td>0.584</td>
<td>8.94</td>
</tr>
<tr>
<td>sample 4</td>
<td>0.325</td>
<td>5.23</td>
</tr>
<tr>
<td>sample 5</td>
<td>0.358</td>
<td>8.51</td>
</tr>
</tbody>
</table>

Since the application has a variety of different $r$ and $S$ values, it is perfect for testing out the benefits of K-Flow in different scenarios. Figure 5.13 compares the throughput between the straightforward integration (Straight), K-Flow integration (K-Flow), and theoretical optimal (Theoretical) on both the local server and the f1.2xlarge instance. The Y-axis represents the actual speedup over the baseline BWA-MEM without FPGA integration. The theoretical optimal solution is calculated by solving the linear programing problem in Equation 5.3 and assumes the number of threads can be a fractional number. In these experiments, we utilize all the available threads in each of our experimental platforms.

The results show two different aspects of the K-Flow on different systems. On the local server, there are many CPU cores and FPGA becomes the bottleneck in the straightforward integration. Therefore the straightforward FPGA integration actually results in slowdowns for all the data samples. An insight is that considering the end-to-end speedup of FPGA integration, including overhead such as data transfer, it is difficult to achieve significant improvement, especially compared to state-of-the-art CPUs. On the other hand, the K-Flow based FPGA integration, BWA-Flow, comes very close to the theoretical limit for a given $r$ and $S$. On f1.2xlarge instance, since there are only 8 cores, the CPU performance becomes the bottleneck, and the impact of FPGA is more significant. As a result, both straightforward and K-Flow implementations achieve good speedups over CPU baseline.
Figure 5.13: Throughput improvement over CPU baseline for various FPGA integrations: straightforward integration (Straight), BWA-Flow (K-Flow) and theoretical optimal (Theoretical).

K-Flow still out-performs straightforward implementation for most cases, however.

On average, K-Flow achieves 94.5% of the theoretical throughput. On our local server, the performance of K-Flow is $1.4 \times$ faster than the straightforward integration across our dataset.

Similar to Figure 5.12, Figure 5.14 shows an example chart of thread assignments with FPGA accelerator during the execution of BWA-Flow, where the total number of threads $P$ is set to 16. This figure demonstrates that besides the beginning and ending stages of BWA-Flow, both the CPU threads and FPGA are fully utilized to achieve close-to-optimal throughput. It also shows that the tasks of $chain2align$ stage are distributed across CPU
threads and the FPGA.

Figure 5.14: Visualization of thread assignment/execution for one sample run of FPGA-integrated BWA-Flow. The total number of CPU threads $P$ is set to 16. Each row in the figure represents the task assignment over time (seconds in X-axis) of an individual CPU thread, except for thread 15 that represents the FPGA thread.

5.7 Discussions

K-Flow offers a new perspective of FPGA acceleration system design—especially when the FPGA speedup is smaller or comparable to multi-core CPU. Using the theoretical analysis in Section 5.4.1, several issues in the accelerator design are discussed in this section.

**Single Fast PE vs. Multiple Slow PEs.** FPGA designers usually need to make decisions between spending resource replicating the same processing element (PE) or increasing the performance of a single PE. From the kernel design point of view, a detailed analysis needs to be conducted to evaluate the two different approaches. But from the host integration point of view, a single fast PE is always better than multiple slow PEs in straightforward integration, if both FPGA designs have the same overall throughput. This is because, in a single fast PE, each CPU thread spends less time waiting for a FPGA task to finish. Figure 5.15 is an example of such a case. This may cause complications in the accelerator
Figure 5.15: Example of the straightforward integration of single fast PE (top) and multiple slow PEs (bottom). The lighter shade represents CPU execution and the darker shade represents FPGA execution. The single PE on the top figure is $4 \times$ faster than each of the four PEs in the bottom figure. The numbers in the labels represent parallel data partitions, and the letter 'a' and 'b' stands for CPU portion and FPGA portion of the data processing respectively. In the single-PE case, the different FPGA tasks need to be sequentialized. In the multi-PE case, four tasks can be executed in parallel. The examples show that the overall throughput of the single-PE implementation is larger because the CPU threads spend less time waiting for the FPGA execution.

design when multi-PE designs can offer better throughput. In K-Flow, there will not be such complications since only the throughput of the FPGA design will affect the overall performance, regardless of the number of PEs.

**Coverage vs. Specificity.** Another question the FPGA designers usually face is whether to accelerate a larger portion of the program or to spend the resource making an existing kernel faster. Using K-Flow, the answer can be easily obtained from Equation 5.4. To simplify the notation, we use the relative speedup with the number of CPU cores $k = \frac{S}{P}$, and we denote the throughput speedup as $Z = \frac{T_{fpga}}{T_{baseline}}$. Then Equation 5.4 can be changed to $Z = \min(\frac{1}{1-r}, 1 + k)$. To help explain the throughput speedup, Figure 5.16 illustrates a 3D plot of the throughput speedup with the kernel ratio $r$ and relative FPGA kernel speedup.
The insight of the plot is that there is an optimal relationship between $k$ and $r$. Given a fixed $r$ value, increasing $k$ passing the optimal point will not result in better throughput, and the same holds true for a fixed $k$ value. The optimal $k$-$r$ curve is plotted in black in Figure 5.16.

Figure 5.16: Illustration of optimal overall program speedup in relation to relative FPGA kernel speedup $k = \frac{S_P}{k}$ and kernel ratio $r$. The optimal $k$-$r$ curve is the smallest $s$ and $r$ value to achieve a certain speedup.

Figure 5.16 can help developers make design decisions trading off coverage acceleration. Depending on where the point on the plot is in relation to the optimal $k$-$r$ curve, the programmer can choose to either expand coverage (increase $r$), or be more specific (increase $k$). Considering the case with 8 CPU cores in the system ($P = 8$), there are two design alternatives: 1) accelerating the entire program with a speedup of $3 \times (k = \frac{3}{8})$ compared to the single core, or 2) accelerating 33% of the program with a speedup of $9 \times (k = \frac{9}{8})$. For the first approach, the overall throughput speedup can be calculated as $1 + \frac{3}{8} = 1.375 \times$. The throughput speedup of the second approach can be calculated as $\frac{1}{1-0.33} = 1.5 \times$. Therefore,
the second approach is better, even though only one-third of the program is accelerated. Moreover, based on the $k$-$r$ curve in Figure 5.16, the optimal $1.5 \times$ speedup is achieved at $k = 0.5$, which means the FPGA kernel speedup is $4 \times$.

### 5.8 Conclusions

This chapter describes K-Flow, an efficient job scheduling system for data streaming applications on CPU-FPGA platforms. With the problem formulation to maximize the system throughput for CPU-FPGA co-execution given the FPGA speedup and the ratio of the accelerated program region, we show that it is possible to achieve the optimal throughput using a dataflow programming model. Thus, we introduce the K-Flow system with a user-friendly programming model and an efficient dynamic task scheduler that can achieve an approximation of the optimal throughput of CPU-FPGA co-optimization. In a case study for genome read alignment, K-Flow achieves on average 94.5% of the optimal throughput and $1.4 \times$ speedup compared to straightforward CPU-FPGA integration, with a manageable amount of programming efforts.

With cluster frameworks such as Spark and Blaze, K-Flow is easy to scale to the datacenter scale. Moreover, the programming model and dynamic scheduling system used in K-Flow is also capable of managing the cluster-level job scheduling in heterogeneous datacenters. We leave these opportunities for future explorations.
CHAPTER 6

Concluding Remarks

This thesis explores the performance and productivity aspect of designing and integrating FPGA-based accelerators into big-data workloads. The solutions are characterized from two different directions of scaling: scale-out and scale-up.

The scale-up system design explores the task-level parallelism of applications. Taking advantage of the flexibility of the FPGA fabrics, the thesis proposes a design methodology that maps tasks to different regions in a single FPGA chip or a multi-FPGA system. Two different domains of applications are used to evaluate this design methodology. The first one is neural microcircuit simulation, which involves modeling of different types of neurons interacting with each other in a network. In the proposed design, dedicated FPGA simulation engines are generated for different neuron populations, which in turn achieves significant performance and energy reduction compared to traditional multi-core CPU systems. A similar pattern can be observed for the second application domain, which is the inferencing in deep convolutional neural networks (DNNs). Much like the neural microcircuits, DNNs consist of different layers—each of which has a unique computation pattern. In this thesis, a framework is designed to efficiently map different DNN layers to a multi-FPGA cluster. The design space exploration of such a framework is formulated and solved using dynamic programming.

The scale-out system design is targeted at the data-level parallelism of big-data applications; this means replicating the same computation for parallel data partitions onto different nodes. In this design space, a runtime system called Blaze is created. Blaze abstracts FPGA-accelerators-as-a-service (FaaS), decouples the FPGA accelerator development and big-data application development, and provides a set of clean programming APIs for big-
data applications to easily access the performance and energy gains of FPGA accelerators. Experiments with four representative big-data applications demonstrate that Blaze greatly reduces the programming efforts, and improves the system throughput from 1.7× to 3×. It is also demonstrated that our FaaS implementation achieves a performance similar to a manual design under the dominant multi-thread scenarios in big-data applications, while our accelerator-centric scheduling achieves close to optimal system throughput.

Many big-data analytic applications actually have both task-level and data-level parallelism. Recognizing this, this thesis combines the scale-up and scale-out design methodologies and proposes a dataflow programing model that pipelines different parallel data partitions for different stages of the computation. Using this model, a system called K-Flow is created. K-Flow includes a user-friendly programming model and an efficient dynamic task scheduler that can achieve an approximation of the optimal throughput of CPU-FPGA co-optimization. In a case study for genome read alignment, K-Flow achieves, on average, a 94.5% of the optimal throughput and 1.4× speedup compared to straightforward CPU-FPGA integration, with a manageable amount of programming efforts.

One key takeaway of this thesis is that although the FPGA architecture is capable of achieving superior performance and energy-efficiency, it takes significant efforts to sustain the benefit at a large system scale. The methodologies proposed in this thesis alleviate such efforts using a systematic approach that includes scaling-up and scaling-out.
REFERENCES


[viv] “Xilinx Vivado HLS.”.


