Deep Learning on FPGAs

by

Griffin James Lacey

A thesis
presented to
the University of Guelph

In partial fulfillment of requirements
for the degree of
Master of Applied Science
in
Engineering

Guelph, Ontario, Canada

ABSTRACT

DEEP LEARNING ON FPGAS

Griffin James Lacey
University of Guelph, 2016

Advisor:
Dr. Graham W. Taylor
Co-Advisor:
Dr. Shawki Areibi

The recent successes of deep learning are largely attributed to the advancement of hardware acceleration technologies, which can accommodate the incredible growth of data sizes and model complexity. The current solution involves using clusters of graphics processing units (GPU) to achieve performance beyond that of general purpose processors (GPP), but the use of field programmable gate arrays (FPGA) is gaining popularity as an alternative due to their low power consumption and flexible architecture. However, there is a lack of infrastructure available for deep learning on FPGAs compared to what is available for GPPs and GPUs, and the practical challenges of developing such infrastructure are often ignored in contemporary work. Through the development of a software framework which extends the popular Caffe framework, this thesis demonstrates the viability of FPGAs as an acceleration platform for deep learning, and addresses many of the associated technical and practical challenges.

Acknowledgements

I would like to thank my advisors, Graham W. Taylor and Shawki Areibi, for their guidance and support throughout this entire process. Their efforts in helping me succeed, both inside and outside of my graduate work, have been truly overwhelming. I feel lucky to have had such incredible advisors.

I can't thank Joel Best enough for his tireless efforts in helping me with the technical challenges of this thesis. Without his expertise, I don't know how I would have retained my sanity dealing with the issues of using bleeding-edge tools. Thanks for always making time to help.

Thank you to all of the lab members, many who have come and gone, who made the long days and late nights more enjoyable. I won't attempt to list names in fear of forgetting someone important, but know that you have all benefited my work in more ways than you may realize. To my University of Toronto friends Jasmina Vasiljevic and Roberto DiCecco, thanks for all of the helpful discussions and advice.

A special thank you to my family for the love and support that is always there when I need it. My parents Lyn and Larry Lacey, my grandparents Mae Benton and Larry Lacey, Sr., my sisters Courtney Lacey and Heather Patel, my brother-in-law Ronak Patel, my nephew Kieran Patel, and my furry relatives Cooper and Casey, thanks for putting up with me this whole time.

Hayley and Wilfrid, thank you for everything, especially the support, perspective, and baked goods.

This work is supported financially by both the Canadian and Ontario governments through the Ontario Graduate Scholarship (OGS) award and the NSERC Canada Graduate Scholarship-Master's Program.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Hardware acceleration has been an integral factor in the advancement of machine learning. The success of a particular sub-class of machine learning techniques, called deep learning, can be largely attributed to the adoption of massively parallel graphics processing units (GPU) running alongside general purpose processors (GPP) for efficient computation of deep neural networks [1]. Moreover, the growing support for deep learning tools in both academia and industry has resulted in a maturing design flow which is accessible to all types of deep learning practitioners [2, 3, 4, 5]. The seamless integration of software (deep learning frameworks) and hardware (high performance computing platforms) has made the rapid prototyping of deep learning models fast and efficient. With the recent popularity of supervised techniques, such as the convolutional neural network (CNN) [6, 7, 8], the emphasis for growth in this field has fallen on developing techniques for hardware acceleration which can accommodate increasing sizes in labelled data sets and model complexity. Most recently, neural networks of increasing depth (as many as 1000 layers) have been shown to improve performance on classification tasks [9]. The future of hardware acceleration for deep learning is far from clear, and recently there has been a spike in interest in investigating alternative high performance computing platforms, such as those with low power or flexible architectures, which may better accommodate the needs of tomorrow.

In this thesis, we investigate the use of field programmable gate arrays (FPGA) as an alternative acceleration platform for deep learning. While this idea is not new, we believe that contemporary work in this field has ignored many important challenges specific to modern deep learning design flows. Therefore, we focus our investigation on the practical challenge of developing a deep learning framework, which uses FPGAs for acceleration. Most importantly, this development attempts to accommodate the conventional deep learning design flow to which practitioners are accustomed. These contributions, along with their enabling technologies, are visualized in Figure 1.1.

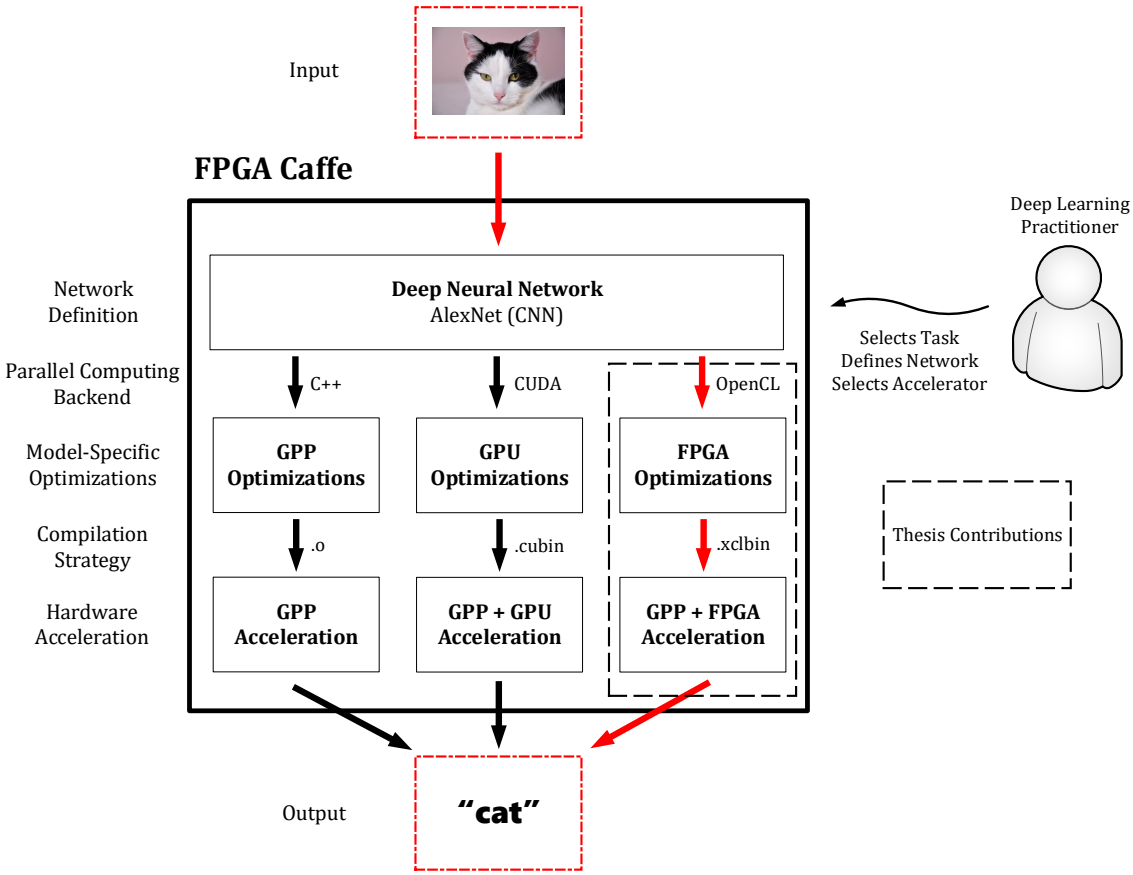


Figure 1.1: This thesis contributes to the development of a practical deep learning framework which extends the popular Caffe tool to include support for FPGA acceleration. This is accomplished through the development of an FPGA-supported backend (OpenCL), model-specific optimizations, and a practical compilation strategy. Validation is done through an empirical performance evaluation of CNN models.

This work reflects on several specific practical challenges related to deep learning on FPGAs:

1. **Design Effort:** Traditional FPGA design tools require hardware-specific knowledge, and are not compatible with software deep learning practices.
2. **Design Iteration:** Compiling designs for FPGAs is very slow compared to GPPs and GPUs, which may be unsuitable for just-in-time compilation methods employed in some deep learning frameworks.
3. **Model Exploration:** Most contemporary works focus only on deployment of deep learning on FPGAs while ignoring training, for which hardware acceleration is even more important.
4. **Multi-Device Parallelism:** Current FPGA technology has been less accommodating of multi-device acceleration schemes compared to GPUs, despite their growing importance for deep learning.
5. **Economic Factors:** Current FPGA technologies are more expensive than GPU alternatives, making it difficult to justify the use of FPGAs for applications which aim to minimize cost.

1.1 Motivations

While FPGAs have shown promising results for deep learning acceleration in contemporary literature, there is a lack of infrastructure supporting deep learning on FPGAs compared to what is available for GPPs and GPUs. Additionally, much of this work on FPGAs builds infrastructure from the ground up, so deep learning practitioners tend to choose more common frameworks which mature through community development. This thesis is motivated by the need for such infrastructure, and the technical and practical challenges of integrating FPGAs into existing deep learning frameworks and design flows. As well, this thesis is enabled by recent developments in the support of the parallel programming framework OpenCL, which provides a sensible technological path between deep learning and FPGAs.

1.2 Objectives

This thesis attempts to answer one main question:

> *Can FPGAs be used as a practical acceleration platform for deep learning?*

Due to the lack of infrastructure, as well as technical and practical challenges related to using FPGAs for deep learning, the answer to this question is unclear. This thesis aims to provide clarity by addressing these challenges through the development of a software framework, through which we are able to demonstrate a practical design flow for deep learning practitioners, and perform an empirical assessment of model-specific performance optimizations for FPGAs. Given that deep learning encompasses a large variety of models and techniques, we focus our investigation toward convolutional neural networks (CNN), as most of the research on FPGA-based deep learning has focused on accelerating CNNs [10]. As well, two important application areas, vision and speech, are presently dominated by CNNs [6, 11].

In addition to these research objectives, this thesis is largely motivated by improving future work in this field. Firstly, by making the proposed framework available to future researchers, it attempts to standardize a methodology in which related work can be performed. Secondly, by describing much of this work from the perspective of a conventional deep learning practitioner who is accustomed to working with GPUs, this thesis acts as a guide for translating those skills and experiences into developing on FPGAs. Lastly, since this work involves the early development of a software framework, we make a strong attempt to guide future improvements and directions of the framework which could not be addressed given the time constraints of this work.

1.3 Contributions

The contributions of this thesis are as follows:

- This thesis contributes to the development of a design flow and practical software framework, FPGA Caffe, for deep learning practitioners to accelerate neural networks using FPGAs.
- This thesis provides an empirical assessment of model-specific optimizations which contribute to the framework. Specifically, this work optimizes deployment of the AlexNet deep convolutional neural network (CNN) [6] on FPGAs and shows competitive performance compared to GPPs and GPUs in terms of execution time and power consumption.

1.4 Relationship to Other Work

This work is largely developed on top of the open-source Caffe deep learning framework, developed by the Berkeley Vision and Learning Center (BVLC) and community contributors [3]. The Caffe project was originally created by Yangqing Jia during his PhD at the University of California, Berkeley. In addition to code, this work draws on inspiration from the structure, function, and motivation of the Caffe project.

The FPGA Caffe project, an extension of Caffe which supports FPGAs, was originally started by Roberto DiCecco during his MASc at the University of Toronto. The development of FPGA Caffe as presented in this work is done in collaboration with Roberto and the Computer Engineering Research Group at the University of Toronto. Details of further development of FPGA Caffe not included in this work are found in "Caffeinated FPGAs: FPGA Framework for Convolutional Neural Networks" by DiCecco et al. [12].

Finally, much of the background and literature review discussed in this work is presented in a related paper which is written to appeal to a much wider audience. For readers looking for a more condensed and digestible discussion on the outlook of deep learning on FPGAs are encouraged to read "Deep Learning on FPGAs: Past, Present, and Future" by Lacey et al. [10].

1.5 Organization

The remainder of this thesis is organized as follows:

Chapter 2 provides background information on deep learning, FPGAs, and the conventional GPU approach used for deep learning acceleration. We begin by providing background information on deep learning, focused on the two models most relevant to this work, the multi-layer perceptron (MLP) and convolutional neural network (CNN). We then discuss background on FPGAs, including a brief overview of FPGA architectures, followed by a discussion of design tools including high-level abstraction tools (HLS) and OpenCL. We conclude with background on deep learning acceleration using GPUs, including a discussion on current deep learning frameworks which use GPU acceleration, as well as a brief overview of GPU parallelism techniques.

Chapter 3 provides a review of contemporary literature, including a timeline of important historical events relevant to this work. This chapter begins with a review of MLPs on FPGAs, and concludes with a review of CNNs on FPGAs.

Chapter 4 provides a discussion of the methodology used in this work. It begins with a high-level discussion of the most popular methods employed for deep learning acceleration, directly comparing this to the GPU approach in terms of framework support and parallelism techniques. We also discuss additional challenges posed by using FPGAs in this context. Next, we discuss the details of the development of the FPGA Caffe framework, including a discussion of the deployment and memory architecture, device kernel design, layer extensions, and design flow.

Chapter 5 provides results of an empirical assessment of model-specific optimizations which contribute to the FPGA Caffe framework. This begins with a brief overview of the experimental setup, then presents the results organized by layer benchmarks, model benchmarks, pipeline layer benchmarks, and power consumption results.

Chapter 6 provides a brief overview of the findings, draws conclusions, and recommends directions for future work.

Chapter 2

Background

The effects of machine learning on our everyday life are far-reaching. Whether you are clicking through personalized recommendations on websites, using speech to communicate with your smart-phone, or using face-detection to get the perfect picture on your digital camera, some form of artificial intelligence is involved. This new wave of artificial intelligence is accompanied by a shift in philosophy for algorithm design. Where past attempts at learning from data involved much "feature engineering" by hand, using expert domain-specific knowledge, the ability to learn composable feature extraction systems automatically from massive amounts of example data has led to ground-breaking performance in important domains such as computer vision, speech recognition, and natural language processing. The study of these data-driven techniques is called deep learning, and is seeing significant attention from two important groups of the technology community: researchers, who are interested in exploring and training these models to achieve top performance across tasks, and application scientists, who are interested in deploying these models for novel, real world applications. However, both of these groups are limited by the need for better hardware acceleration to accommodate scaling beyond current data and algorithm sizes.

The current state of hardware acceleration for deep learning is largely dominated by using clusters of

graphics processing units (GPU) as general purpose processors (GPGPU) [1]. GPUs have orders of magnitude more computational cores compared to traditional general purpose processors (GPP), and allow a greater ability to perform parallel computations. In particular, the NVIDIA CUDA platform for GPGPU programming is most dominant, with major deep learning tools utilizing this platform to access GPU acceleration [2, 3, 4, 5]. More recently, the open parallel programming standard OpenCL has gained traction as an alternative tool for heterogeneous hardware programming, with interest from these popular tools gaining momentum. OpenCL, while trailing CUDA in terms of support in the deep learning community, has two unique features which distinguish itself from CUDA. First is the open source, royalty-free standard for development, as opposed to the single vendor support of CUDA. The second is the support for a wide variety of alternative hardware including GPUs, GPPs, field programmable gate-arrays (FPGA), and digital signal processors (DSP).

The imminent support for alternative hardware is especially important for FPGAs, a strong competitor to GPUs for algorithm acceleration. Unlike GPUs, these devices have a flexible hardware configuration, and often provide better performance per watt than GPUs for subroutines important to deep learning, such as sliding-windows computation [13] and dense matrix multiplication [14]. However, programming of these devices requires hardware-specific knowledge that many researchers and application scientists may not possess, and as such, FPGAs have been often considered a specialist architecture. Recently, FPGA tools have adopted software-level programming models, including OpenCL, which has made them a more attractive option for users trained in mainstream software development practices.

For researchers considering a variety of design tools, the selection criteria is typically related to having user-friendly software development tools, flexible and upgradeable ways to design models, and fast computation to reduce the training time of large models. Deep learning researchers will benefit from the use of FPGAs given the trend of higher abstraction design tools which are making FPGAs easier to program, the reconfigurability which allows customized architectures, and the large degree of parallelism which will accelerate execution speeds. For application scientists, while similar tool-level preferences exist, the emphasis for hardware selection is to maximize performance per watt of power consumption, reducing costs for large-scale operations. Deep learning application scientists will benefit from the use of FPGAs given the strong performance per watt that typically accompanies the ability to customize the architecture for

a particular application. FPGAs serve as a logical design choice which appeal to the needs of these two important audiences, herein referred to as deep learning practitioners.

The following chapter discusses the background information relevant to this work. This is accomplished by first introducing deep learning, and providing context for the modern deep learning paradigm within the history of artificial intelligence. To narrow the scope of this discussion, we then introduce the two specific types of neural networks which are most relevant to FPGA accleration, multi-layer perceptrons (MLP) and convolutional neural networks (CNN). This section also includes a brief description of several important CNN layer types which this work implements for experimentation. Next, we introduce FPGAs, and discuss two important design tools for FPGAs which are relevant to this work, high-level synthesis and OpenCL. Finally, we discuss the current state of conventional deep learning acceleration using GPUs, including the current design flows for acceleration, and the commonly employed methods of parallelism used for design. The conventional approach is necessary to provide context for comparison with the unconventional FPGA approach introduced in this work.

2.1 Deep Learning

Traditional approaches to artificial intelligence focused on using computation to solve problems analytically, requiring explicit knowledge about a given domain [15]. For simple problems, this approach was adequate, as the programs engineered by hand were small, and domain experts could carefully transform the modest amount of raw data available into useful representations for learning. However, advances in artificial intelligence created interest in solving more complex problems, where knowledge is not easily expressed explicitly. Expert knowledge about problems such as face recognition, speech transcription, and medical diagnosis is difficult to express formally, and conventional approaches to artificial intelligence failed to account for the implicit information stored in the raw data. Moreover, tremendous growth in data acquisition, particularly of the unstructured type, as well as storage means that using this implicit information is more important than ever. Recently, these types of applications are seeing state-of-the-art performance from a class of techniques called deep learning, where this implicit information is discovered

automatically by learning task-relevant features from raw data. Interest in this research area has led to several recent reviews [16, 17, 18].

The field of deep learning emerged around 2006 after a long period of relative disinterest around neural networks research. Interestingly, the early successes in the field were due to unsupervised learning - techniques that can learn from unlabeled data. Specifically, unsupervised learning was used to "pre-train" (initialize) the layers of deep neural networks, which were thought at the time to be too difficult to train with the usual methods, i.e. gradient backpropagation. However, with the introduction of GPGPU computing and the availability of larger datasets towards the end of the 2000's and into the current decade, focus has shifted almost exclusively to supervised learning. In particular, there are two types of neural network architectures that have received most of the attention both in research and industry. These are multi-layer perceptrons (MLP) and convolutional neural networks (CNN). Essentially all of the research on FPGA-based deep learning has focused on one of these architectures.

Before describing any specific architecture, however, it is worth noting several characteristics of most deep learning models and applications that, in general, make them well-suited for parallelization using hardware accelerators.

Data parallelism – The parallelism inherent in pixel-based sensory input (e.g. images and video) manifests itself in operations that apply concurrently to all pixels or local regions. As well, the most popular way of training models is not by presenting it with a single example at a time, but by processing "mini-batches" of typically hundreds or thousands of examples. Each example in a mini-batch can be processed independently.

Model parallelism – These biologically-inspired models are composed of redundant processing units which can be distributed in hardware and updated in parallel. Recent work on accelerating CNNs using multiple GPUs has used very sophisticated strategies to balance data and model-based parallism such that different parts of the architecture are parallelized in different, but optimal ways [19].

Pipeline Parallelism – The feed-forward nature of computation in architectures like MLPs and CNNs

means that hardware which is well suited to exploit pipeline parallelism (e.g. FPGAs) can offer a particular advantage. While GPPs and GPUs rely on executing parallel threads on multiple cores, FPGAs can create customized hardware circuits which are deeply pipelined and inherently multithreaded.

2.1.1 Multi-Layer Perceptrons

Simple feed-forward deep networks are known as multi-layer perceptrons (MLP), and are the backbone of deep learning [20]. To describe these models using neural network terminology, we refer to the examples fed to these models as *inputs*, the predictions produced from these models as *outputs*, each modular sub-function as a *layer* with *hidden layers* referring to those layers between the first (input) layer and last (output) layer, each scalar output of one of these layers as a *unit* (analogous to a neuron in the biological analogy of these models), and each connection between units as a *weight* (analogous to a synapse), which define the function of the model as they are the parameters that are adjusted during training [18]. Collections of units are sometimes referred to as *features*, as to draw similarities to the traditional idea of features in conventional machine learning, which, in the past, were most often designed by domain experts. To prevent the entire network from collapsing to a linear transformation, each unit applies an element-wise nonlinear operation to its input, with the most popular choice being the rectified linear unit (ReLU).

2.1.2 Convolutional Neural Networks

Deep convolutional neural networks (CNN) are currently the most popular deep learning architecture, especially for pixel-based visual recognition tasks. More formally, these networks are designed for data that has a measure of spatial or temporal continuity. This inspiration is drawn largely on work from Hubel and Wiesel, who described the function of a cat's visual cortex as being sensitive to small sub-regions of the visual field [21]. Commonly, spatial continuity in data is found in images where a pixel at location (i, j) shares similar intensity or color properties to its neighbours in a local region of the image.

While CNN architectures have many customizable parameters that make them unique to each other, CNNs are typically composed of predictable combinations of a few important layer types. Convolution layers, pooling layers, and fully connected layers are fundamental to CNNs and found in virtually all CNNs. In comparison to MLPs, these layers are constructed as a 2D arrangement of units called *feature maps*. Convolution layers, analogous to the linear feature extraction operation of MLPs, are parameterized by learnable filters, which have local connections to a small receptive field of the input feature map and shared at all locations of the input. Feature extraction in a CNN amounts to convolution with these filters. Pooling layers apply a simple reduction operation (e.g. a max or average) to local regions of the feature maps. This reduces the size of the feature maps, which is favorable to computation and reducing parameters, but also yields a small amount of shift-invariance. Finally, in recognition applications CNNs typically apply one or more fully connected layers (the same layers used in MLPs) towards the output layer in order to reduce the spatially and/or temporally organized information in feature maps to a decision, such as a classification or regression. While CNNs could be composed of any number of different configurations of these or other layer types, we focus our discussion on the layer types that are used in the AlexNet CNN, as this is the model chosen to be implemented in this work. The details of these layers are discussed below.

Fully Connected Layers

While CNN architectures consist mostly of convolution and pooling layers, fully connected layers, which are the same layers used in MLPs, are used near the output layer to capture global context and model high-level concepts. A fully connected layer connects all units in the previous layer to all units in the current layer, and can be performed using a matrix multiplication and addition of a bias term. Consider a neural network with L hidden layers expressed in matrix form:

$$\mathbf{h}^{(l)} = \sigma^{(l)}(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}) \tag{2.1}$$

where we define $\mathbf{h}^{(l)}$ as the output of the hidden layer l, $\sigma^{(l)}$ as an element-wise non-linear activation, $W^{(l)}$

as a weight matrix, $\mathbf{h}^{(l-1)}$ as the output of the previous hidden layer, and $\mathbf{b}^{(l)}$ as a bias. The activation function, chosen by the architect of the model, is discussed in more detail below.

Convolution Layers

In discrete mathematics, the convolution of a 1D signal f with a signal g is defined in Equation 2.2,

$$o[n] = f[n] * g[n] = \sum_{u=-\infty}^{\infty} f[u]g[n-u] \tag{2.2}$$

where n & u are discrete variables. However, we can also extend this definition to include convolution in a second dimension, as required for pixel-based convolution, as follows:

$$o[m,n] = f[m,n] * g[m,n] = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f[u,v]g[m-u,n-v] \tag{2.3}$$

where m and n index the two dimensions of the original signal, and u, v index the second signal. However, when considering convolution applied to pixel-based input, such as images, we no longer have signals defined on the interval of $(-\infty, \infty)$; the signals are now constrained to finite numbers, which are often the sizes of the images and associated filters. In CNNs, these convolutional filters allow features in sub-regions of the image to be learned. These filters are treated as parameters to be learned through back propagation - similar to how traditional weights W and biases b in MLPs are used. Each convolution layer often has multiple filters associated with it, and thus produces multiple output feature maps. Mathematically, convolution layers are expressed very similarly to fully connected layers:

$$\mathbf{h}^{(l)} = \sigma^{(l)}(\mathbf{W}^{(l)} * \mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}) \tag{2.4}$$

where instead of matrix multiplication, the weights $\mathbf{W}^{(l)}$ and input $\mathbf{h}^{(l-1)}$ are convolved, denoted by $*$. In terms of computational power, these convolution layers are often the most expensive to perform due to the number of multiplications and additions that need to be computed. For hardware acceleration, the main focus is to efficiently perform the convolution operation.

ReLU Activation

Similar to MLPs, a non-linear activation function is applied to the linear output of every convolution to prevent the network from collapsing to a single layer. While traditional neural networks have favoured the tanh and sigmoid (logistic) activation functions, many state-of-the-art networks are now using rectified linear units (ReLU) [22], as seen in Equation 2.5:

$$y = max(0, x) \tag{2.5}$$

where the activation y is strictly positive definite. While not typically considered a layer of a neural network, we will refer to ReLU as a layer to provide convenience when discussing the implementation specifics. This activation avoids some of the pitfalls of neural networks, including when their units are acting in the saturated region. Differences between the tanh, sigmoid, and ReLU activation characteristics can be seen in Figure 2.1.

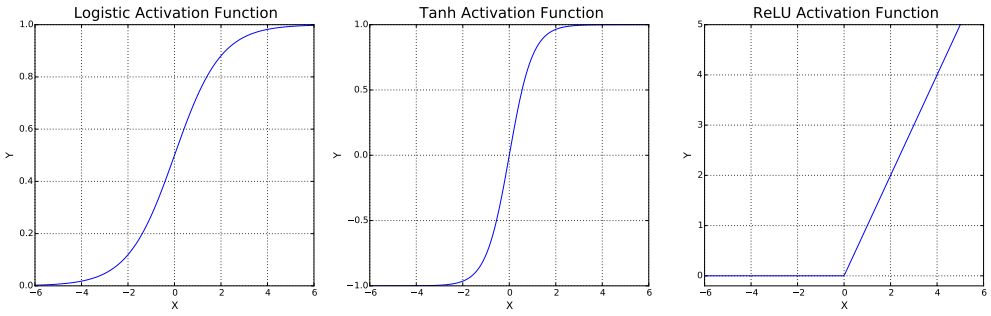


Figure 2.1: Common activation functions used in neural networks.

Pooling Layers

Pooling layers are commonly used following convolution and non-linear activation layers. In pooling, local regions of the previous layer are replaced with statistics that summarize the neighbouring outputs, meaning the size of every input map is spatially reduced. The most common type of pooling is max pooling, where the maximum value of a rectangular a neighbourhood is reported. Consider a pooling layer with a square input matrix of size ${N_{in}}^2$, which produces a square output matrix of size ${N_{out}}^2$. Assuming no overlapping of filters, the input is divided into pooling regions $P_{i,j}$ where

$$P_{i,j} \subset \{1, ..., N_{in}\}^2 \text{ for each } (i, j) \in \{1, ..., N_{out}\}^2. \tag{2.6}$$

Max pooling is then seen as a *max* function applied element-wise to a given pooling region, and tiled across an entire input image as seen in Equation 2.7.

$$\text{output}_{i,j} = \max_{(k,l) \in P_{i,j}} \text{input}_{k,l} \tag{2.7}$$

Determining the output dimensions is a function of the input dimensions, pooling filter size and stride. Consider the case for a square input matrix and filter:

$$N_{out} = \frac{N_{in} - F}{S + 1} \tag{2.8}$$

where F is the pooling filter size, and S is the stride. While most common pooling is done on square dimensions, extending this relationship to non-square dimensions is trivial. In addition to reducing dimensionality, pooling aides in making a representation invariant to small shifts or translations in the input, which is useful for applications that care more about the presence of a feature than the exact spatial

position of that feature. In practice, pooling has computational conveniences such as handling variable sized inputs, and reducing memory requirements for parameter storage.

Local Response Normalization Layers

While in theory, no normalization method is needed to achieve convergence when training neural networks, it is common practice to use normalization techniques to help speed up the rate of convergence. Since the distribution of layer inputs changes during training, the use of high learning rates can be troublesome. One solution to this problem is the use of normalization layers, which normalize layer inputs and allow higher learning rates to be used, which empirically helps speed up training. As used in the AlexNet CNN architecture, local response normalization layers (LRN) normalize across adjacent channels (input examples). By denoting the activity of a unit $a^i_{x,y}$ as the application of kernel i to position (x, y) which is then applied through a non-linear activation. The normalized response $b^i_{x,y}$ is given by

$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{max(0,i-n/2)}^{min(N-1,i+n/2)} ({a^i_{x,y}}^2) \right)^{\beta} \tag{2.9}$$

where N is the total number of kernels per layer, the summation is performed over n adjacent feature maps at the same spatial location, and k, n, α, and β are hyper-parameters [6]. Intuitively, these layers enforce a local competition between adjacent units, encouraging neighbouring units to compete for representing significant features. While this layer type was important in the AlexNet model, it has now been largely ignored in favour of newer techniques such as batch normalization [23].

2.2 FPGAs

Traditionally, when evaluating hardware platforms for acceleration, one must inevitably consider the trade-off between flexibility and performance. At one end of the spectrum, general purpose processors (GPP) provide a high degree of flexibility and ease of use, but perform relatively inefficiently. These platforms tend to be more readily accessible, can be produced cheaply, and are appropriate for a wide variety of uses and reuses. At the other end of the spectrum, application specific integrated circuits (ASIC) provide high performance at the cost of being inflexible and more difficult to produce. These circuits are dedicated to a specific application, and are expensive and time consuming to produce.

FPGAs serve as a compromise between these two extremes. They belong to a more general class of programmable logic devices (PLD) and are, in the most simple sense, a reconfigurable integrated circuit. As such, they provide the performance benefits of integrated circuits, with the reconfigurable flexibility of GPPs. At a low-level, FPGAs can implement sequential logic through the use of flip-flops (FF) and combinational logic through the use of look-up tables (LUT). Modern FPGAs also contain hardened components for commonly used functions such as full processor cores, communication cores, arithmetic cores, and block RAM (BRAM). In addition, current FPGA trends are tending toward a system-on-chip (SoC) design approach, where ARM coprocessors and FPGAs are commonly found on the same fabric. The current FPGA market is dominated by Xilinx and Altera, accounting for a combined 85 percent market share [24]. In addition, FPGAs are rapidly replacing ASICs and application specific standard products (ASSP) for fixed function logic. The FPGA market is expected to reach the $10 billion mark in 2016 [24].

For deep learning, FPGAs provide an obvious potential for acceleration above and beyond what is possible on traditional GPPs. Software-level execution on GPPs rely on the traditional Von Neumann architecture, which stores instructions and data in external memory to be fetched when needed. This is the motivation for caches, which alleviate much of the expensive external memory operations [24]. The bottleneck in this architecture is the processor and memory communication, which severely cripples GPP performance, especially for the memory-bound techniques frequently required in deep learning. In comparison, the programmable logic cells on FPGAs can be used to implement the data and control path found in common logic functions, which do not rely on the Von Neumann architecture. They are also capable of exploiting

distributed on-chip memory, as well as large degrees of pipeline parallelism, which fit naturally with the feed-forward nature deep learning methods. Modern FPGAs also support partial dynamic reconfiguration, where part of the FPGA can be reprogrammed while another part of the FPGA is being used. This can have implications for large deep learning models, where individual layers could be reconfigured on the FPGA while not disrupting ongoing computation in other layers. This would accommodate models which may be too large to fit on a single FPGA, and also alleviate expensive global memory reads by keeping intermediate results in local memory.

Most importantly, when compared to GPUs, FPGAs offer a different perspective on what it means to accelerate designs on hardware. With GPUs and other fixed architectures, a software execution model is followed, and structured around executing tasks in parallel on independent compute units. As such, the goal in developing deep learning techniques for GPUs is to adapt algorithms to follow this model, where computation is done in parallel, and data interdependence is ensured. In contrast, the FPGA architecture is tailored for the application. When developing deep learning techniques for FPGAs, there is less emphasis on adapting algorithms for a fixed computational structure, allowing more freedom to explore algorithm level optimizations. Techniques which require many complex low-level hardware control operations which cannot be easily implemented in high-level software languages are especially attractive for FPGA implementations. However, this flexibility comes at the cost of large compile (synthesis) times, which is often problematic for researchers who need to quickly iterate through design cycles.

In addition to compile time, the problem of attracting researchers and application scientists, who tend to favour high-level programming languages, to develop for FPGAs has been especially difficult. While it is often the case that being fluent in one software language means one can easily learn another, the same cannot be said for translating skills to hardware languages. The most popular languages for FPGAs have been Verilog and VHDL, both examples of hardware description languages (HDL). The main difference between these languages and traditional software languages, is that HDL is simply describing hardware, whereas software languages such as C are describing sequential instructions with no need to understand hardware level implementation details. Describing hardware efficiently requires a working knowledge of digital design and circuits, and while some of the low level implementation decisions can be left to automatic synthesizer tools, this does not always result in efficient designs. As a result, researchers and application

scientists tend to opt for a software design experience, which has matured to support a large assortment of abstractions and conveniences that increase the productivity of programmers. These trends have pushed the FPGA community to now favour design tools with a high-level of abstraction.

2.2.1 High-Level Synthesis

Both Xilinx and Altera have favoured the use of high-level design tools which abstract away many of the challenges of low level hardware programming. These tools are commonly termed high-level synthesis (HLS) tools, which translate high-level designs into low-level register-transfer level (RTL) or HDL code. A good overview of HLS tools is presented in [24], where they are grouped into five main categories: model-based frameworks, high-level language based frameworks, HDL-like languages, C-based frameworks, and parallel computing frameworks (i.e. CUDA/OpenCL). While it is important to understand these different types of abstraction tools, this work focuses on parallel computing frameworks, as they provide the most sensible path to join deep learning and FPGAs.

2.2.2 OpenCL

OpenCL is an open source, standardized framework for algorithm acceleration on heterogeneous architectures. As a C-based language (C99), programs written in OpenCL can be executed transparently on GPPs, GPUs, DSPs, and FPGAs. Similar to CUDA, OpenCL provides a standard framework for parallel programming, as well as low-level access to hardware. While both CUDA and OpenCL provide similar functionality to programmers, key differences between them have left most people divided. Since CUDA is the current choice for most popular deep learning tools, it is important to discuss these differences in detail, in the interest of demonstrating how OpenCL could be used for deep learning moving forward.

The major difference between OpenCL and CUDA is in terms of ownership. CUDA is a proprietary framework created by the hardware manufacturer NVIDIA, known for manufacturing high performance

GPUs. OpenCL is open-source, royality-free, and is maintained by the Khronos group. This gives OpenCL a unique capability compared to CUDA: OpenCL can support programming a wide variety of hardware platforms, including FPGAs. However, this flexibility comes at a cost, where all supported platforms are not guaranteed to support all OpenCL functions. In the case of FPGAs, only a subset of OpenCL functions are currently supported. While a detailed comparison of OpenCL and CUDA is outside the scope of this work, performance of both frameworks has been shown to be very similar for given applications [25].

Beginning in late 2013, both Altera and Xilinx started to adopt OpenCL SDKs for their FPGAs [26, 27]. This move allowed a much wider audience of software developers to take advantage of the high-performance and low power benefits that come with designing for FPGAs. Conveniently, both companies have taken similar approaches to adopting OpenCL for their devices. The key to understanding OpenCL is to understand how devices are organized, memory is organized, and how programs are executed. This amounts to understanding the OpenCL platform, memory, and execution model.

The OpenCL platform model defines how hardware is structured within the OpenCL standard, and is designed to accommodate heterogeneous accelerators, where one host device is connected to one or more accelerators (e.g. FPGA, GPU) called compute devices. Each compute device is broken down into multiple compute units, which are further broken down into processing elements. Computation on a compute device occurs on the processing elements. These terms are purposefully ambiguous in a way such that compute devices of variable architectures can be described with useful consistency. A diagram of the OpenCL platform model is seen in Figure 2.2. It should be noted that while the OpenCL standard can deal with multiple compute devices, this work addresses the case of a single host interacting with a single compute device (FPGA).

The OpenCL memory model describes the structure and behaviour of memory within the OpenCL standard. It includes memory that is exposed to the host and compute devices, and how memory is viewed from each devices. The following list summarizes the memory model structure.

- **Host memory** is the region that is only visible and accessible only to the host processor. OpenCL devices cannot access this memory region, and as such, any data needed by the device must be

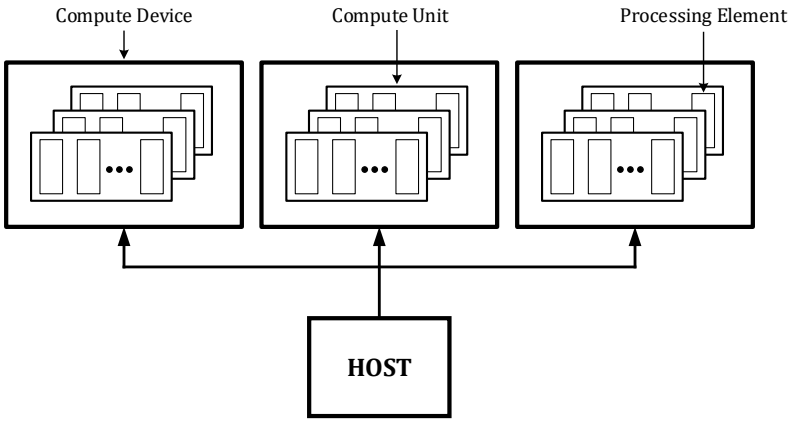


Figure 2.2: The OpenCL platform model describes a host device (i.e. GPP) connected to one or more compute devices (e.g. FPGA, GPU) which are broken down into compute units, and processing elements [28].

transferred to a device-accessible region such as global memory. In FPGA systems, host memory refers to any memory connected only to the host.

- **Global memory** is the region that is visible and accessible by both the host processor and device. While the host controls the access of this memory region prior to kernel execution, once kernel execution begins this control is handed off to the device. The use of *global* in this context refers to the property that all work-items in all work-groups have access to this memory region. In FPGA systems, global memories are usually off-chip banks connected to the FPGA, such as DDR3 SDRAM.

- **Constant memory** is a region of global memory that is read-only to the device, and therefore immutable during kernel execution. In FPGA systems, this memory region is the same as global memory.

- **Local memory** is the device region that is only visible to work-items in the same work-group. In FPGA systems, local memory is implemented with block RAM of the FPGA fabric.

- **Private memory** is the device region that is only visible to singular work-items. In FPGA systems, private memory is implemented using registers of the FPGA fabric.

Finally, the execution model describes how computation is performed under the OpenCL standard. While a full description of the this model not necessary to understand this work, the fundamental structure is described in the following list:

- **Context** is the environment within which the desired computation is performed, and includes information describing the devices, memory, commands, and schedule for execution.
- **Program** consists of a set of device kernels and other functions, which act as a dynamic library of functions to perform the desired task.
- **Device kernel** is the function implementation which is executed on a compute device. While commonly referred to as simply a *kernel*, we use the term *device kernel* in this work is differentiate from the use of the term *kernel* in machine learning.
- **Work-group** is a collection of work-items which execute the same device kernel instance.
- **Work-item** is the basic unit of work in OpenCL, and is a collection of parallel executions of a particular device kernel executing on a compute device.

It should be noted that, while the FPGA design tools used in this work fully support OpenCL, the methodology in which OpenCL is used is slightly different than common OpenCL devices such as GPUs and GPPs. These important differences are described in Chapter 4 as they become relevant to the discussion.

2.3 Deep Learning Acceleration using GPUs

In practice, most conventional implementations of neural networks use graphics processing units (GPU) for acceleration. Though originally intended for graphics applications, the use of GPUs for general purpose computation, GPGPU, is growing in popularity. Conveniently, the hardware requirements of deep learning and graphics applications are very similar. Firstly, deep learning requires high memory bandwidth to store large buffers of gradients, activations, and parameters. Graphics applications require high memory

bandwidth to store information relating to object textures and colours. Secondly, deep learning requires a high degree of parallelism to process independent neurons concurrently. Graphics applications require a high degree of parallelism to process pixels and objects in images concurrently. Deep learning applications also do not typically require sophisticated branching or control logic, making them a good fit for GPUs [29].

2.3.1 GPU-Based Deep Learning Frameworks

The large majority of deep learning practitioners use one or more open-source deep learning frameworks to design and test their models, with these frameworks commonly accessing GPU acceleration through a CUDA backend. From the perspective of the practitioner, complex deep learning and hardware acceleration techniques can be accessed through a high-level language such as Python or C++. A block diagram of abstraction hierarchy for deep learning practitioners is seen in Figure 2.3.

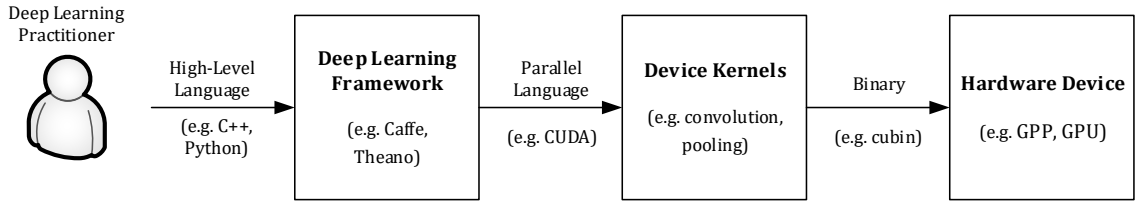


Figure 2.3: GPU-based deep learning framework abstraction hierarchy for deep learning practitioners.

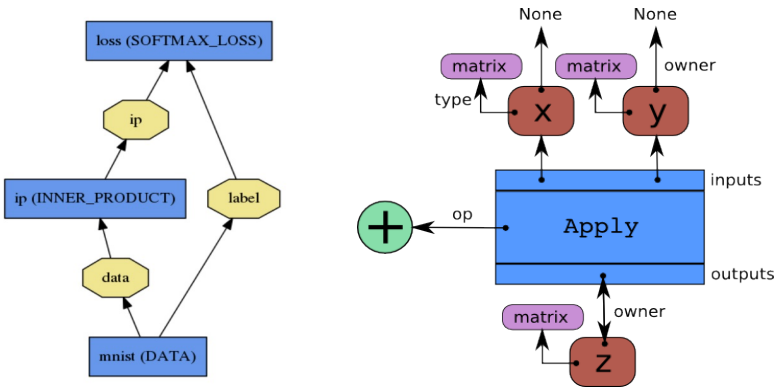
From Figure 2.3, we see that the interaction between the deep learning practitioner and the deep learning framework is a high-level programming language. The programming language is dependent on the choice of deep learning framework, however, many frameworks include bindings for multiple high-level language interfaces. A brief overview of these popular deep learning frameworks, including which languages and backends are supported, is shown in Table 2.1. It should be noted that this table describes only the original projects, and does not include details on libraries built on top of these frameworks that provide additional abstractions (e.g. the Lasagne, Keras, or Blocks libraries for Theano).

Table 2.1: Overview of supported high-level languages in popular deep learning frameworks

Tool	Core Language	Bindings	CUDA	OpenCL
Caffe [3]	C++	Python MATLAB	Yes	Unofficial Support
Torch [5]	Lua	-	Yes	Unofficial Support
Theano [4]	Python	-	Yes	Official Support
TensorFlow [30]	C++	Python	Yes	No Support

At the backend of the deep learning framework, a parallel computing language is used to express the computationally expensive routines in a structure which allows access to parallel computation. Here, we define these routines written in a parallel computing language for hardware acceleration as *device kernels*. As seen in Table 2.1, most popular deep learning frameworks use CUDA as the backend for GPU acceleration. However, many tools are starting to support OpenCL as an alternative. While this support is compatible with many OpenCL devices, FPGAs have unique OpenCL characteristics which means that these tools would not support FPGAs out-of-the-box simply because they support OpenCL. This issue, a main concern of this thesis, it addressed in further detail in Chapter 4. Since designing these device kernels for high performance can be difficult, most deep learning practitioners should attempt to structure their workflow to reuse existing device kernels, rather than designing new ones. When these device kernels are needed for needed for execution, they are compiled to executable code (e.g. `cubin`) which contains instructions to structure the hardware to perform the required task. This executable code can be compiled either online (run time) or offline (compile time), and is loaded to the compute device for execution at run time.

While Figure 2.3 shows how many implementation details of deep learning frameworks are abstracted away from the practitioner, these details become important for understanding how to design efficiently for hardware. At a lower level, most deep learning frameworks work fundamentally the same, by creating a computational graph to describe a deep learning model. In Caffe, for example, computational graphs are described through the connections of neural network layers (e.g. convolution, pooling) where each layer contains a function defining the forward (output computation) and backward (gradient computation) activity of the layer. In Theano, however, computational graphs are described through symbolic mathematical expressions, which are much more general than just neural networks. These graphs use different types of nodes (e.g. apply, variable, and operation) to represent mathematical expressions, such as convolution,



(a) Caffe computational graph[1] (b) Theano computational graph[2]

Figure 2.4: Computational graphs as represented in Caffe (left) by connections of layers, and Theano (right) by connections of nodes which describe more general mathematical computation (e.g. apply, variable, and operator nodes).

which can be used to efficiently build neural networks. These graphs are visualized in Figure 2.4.

Despite the differences in framework structure, the hardware acceleration methodology is very similar across frameworks. Since computational graphs are modular, device code can be extracted from modular components of the graph. Typically, device code is organized by the layer structure of the neural network. For example, consider a convolution layer implemented in Caffe and Theano. In Caffe, a convolution layer is represented by a node in the computational graph, and the device code for convolution is written as a member function of that layer. In Theano, convolution is represented as an operator, and so Theano must intelligently replace convolution operators with the respective device code during an optimization phase. In addition, the input and output variables would have to be declared as specific types (shared variables) such that they are located in memory on the hardware device. While we limit our discussion to Caffe and Theano as examples, these ideas extend to encompass most popular deep learning frameworks. To remain agnostic to framework specific terminology, we define each section of device code to be accelerated on the hardware device as a *module*.

[1] `http://caffe.berkeleyvision.org/tutorial/net_layer_blob.html`
[2] `http://deeplearning.net/software/theano/extending/graphstructures.html`

2.3.2 GPU Parallelism

In deep learning frameworks using GPU acceleration, the modules representing the computationally expensive operations (e.g. convolution, pooling, etc...) are organized into CUDA subroutines and compiled either offline or online to be executed on the GPU. While CUDA is a software programming model, some knowledge about the underlying GPU hardware architecture and parallel programming is required for designing highly efficient device kernels. To ensure consistency, we use OpenCL terminology when describing parallel architectures, such as work-item (CUDA thread), work-group (CUDA thread block), local memory (CUDA shared memory), and private memory (CUDA local memory). Despite the differences in terminology, these ideas can be applied to both OpenCL and CUDA designs. Figure 2.5 shows the structure for accelerating computational graphs on GPUs, including a visualization of GPU parallelism.

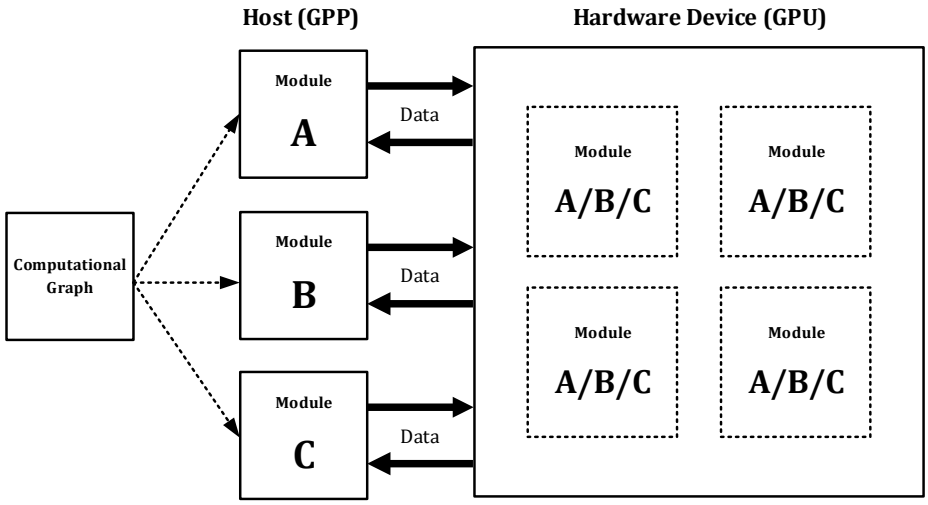


Figure 2.5: For GPU acceleration, computational graphs are broken down into modules, which can be replicated on the GPU to maximize throughput. On GPUs, only one unique module can be executing at any given time, and each module exchanges data with the GPU between executions.

Data Parallelism

Typically, efficient GPU kernel design for deep learning is achieved through data parallelism, which is often inherent in the pixel-based sensory input (e.g. images and video) relevant to CNNs. The pixels can be operated on concurrently, as independent local regions. In CUDA, this parallelism is achieved through tiling, where the problem space is broken down as a grid model. As a grid, the input is partitioned into small, independent chunks that can be executed in parallel. Each chunk, which will execute as a work-item, operates on a small enough portion of the data such that it will fit in private memory, reducing the amount of high latency global memory reads. As such, each kernel operation is described from the point of view of a single work-item, and this work-item is duplicated enough times to cover the entire problem space. While other parallel programming techniques are available, this approach is often best for deep learning, given the satisfaction of the basic assumption of data independence. This idea of low-level data parallelism within a single image can be extended further to include high-level parallelism across groups of images. Since input examples are often independent, processing "mini-batches" of typically hundreds or thousands of input examples is common practice. GPGPU is fundamentally a SIMD approach, as each kernel performs a single operation (e.g. convolution, pooling, etc...) on multiple pieces of the input data.

Model Parallelism

Where data parallelism uses the same model in every partition but uses different parts of the input data, model parallelism uses the same input data but splits the model across partitions. Model parallelism can be especially useful for accommodating large neural network sizes, as well as performing efficient model search by training multiple networks at the same time and averaging the results. Recent work on multi-GPU acceleration has used techniques which balance data and model parallelism across different layers of the network [19].

2.4 Summary

In this chapter, we discussed background information on deep learning, FPGAs, and the conventional GPU-based approach for deep learning acceleration. We focused our deep learning background information on CNNs, while our FPGA discussion focused on OpenCL as a high level design tool. The background on the conventional GPU-based approach for deep learning acceleration focused on which frameworks are commonly used, how they differ, and how parallelism is commonly achieved with GPU acceleration. This discussion was intended to give an overview on the current state of deep learning hardware acceleration, so as to better understand how FPGAs may fit into this paradigm. We notice that no commonly used deep learning frameworks currently support FPGAs.

Chapter 3

Literature Review

Since most contemporary work related to deep learning on FPGAs has focused on MLP and CNN architectures, we focus our literature review to these two architectures. This chapter provides a review of both MLP and CNN implementations using FPGAs and is written mostly chronologically, outlining important historical events and leading into the current state-of-the-art.

3.1 Multi-Layer Perceptrons on FPGAs

The first FPGA implementations of neural networks began appearing in the early 1990's, with the first implementation credited to Cox et al. in 1992 [31]. While a great change in neural network understanding, computer hardware, and terminology has changed since this time, the basic functionality of these networks has remained the same. What began as connectionist classifiers has now become referred to as artificial neural networks or multi layer perceptrons, and FPGA implementations of these networks were, at first, strongly limited by the logic density of FPGAs at the time. Still, several strong efforts were seen before

the turn of the century, including [32, 33, 34, 35]. A common theme in these early implementations was to attempt to increase the density of these implementations, and allow for scalable, reconfigurable architectures of feed-forward neural networks. During this time, dense hardened multiply-accumulate (MAC) units were not available in most FPGAs. As a result, the heavy multiplication requirements of these networks were resource expensive on FPGAs, as they had to be realized using the generic FPGA fabric. For a review of ANN progress during the first decade of existence, readers are encouraged to read the work of Zhu et al. circa 2003 [36]. Another similar review conducted by Omondio et al. in 2006, focused on specific papers that, to date, marked important progress towards the goal of FPGA-based neurocomputers [37]. This includes an in-depth look at multi-layer MLPs [38], accelerating back propagation [39, 40], and the impact of arithmetic representation for MLPs [41]. The next decade of progress for FPGA implementations continued in a similar trend, with special interests in efficiency as the capabilities of modern FPGAs grew, and applications of neural networks began to see adoption in a more mainstream sense. Vitabile et al. demonstrated efficient MLP designs in [42], while designs for low cost FPGAs began to emerge, such as that presented by Ordonez et al. [43]. For a more detailed look at progress during this second decade of existence, readers are encouraged to look at the work of Misra et al. [44]. Of additional interest to readers of this survey may be the treatment of both alternative network structures, and alternative learning schemes, topics of which are outside the scope of this thesis.

The third decade of existence brings us to our current state, which inches closer to the goal of realizing complete deep learning systems on FPGAs. Up until now, however, learning had been mostly considered a separate problem from evaluation (forward propagation with no learning) of the model. As such, most designs involved off-chip learning, where the weights were exported and fixed within the FPGA, greatly reducing the complexity of the design. While this is acceptable for many applications, the goal of deep learning on FPGAs is one where learning is done on-chip, something that only became possible for large networks on single FPGAs somewhat recently. Readers are encouraged to investigate a survey of on-chip learning realizations for FPGAs by Lakshmi et al. for a more detailed discussion [45]. One of the first efforts in realizing this goal is done by Gomperts et al., who describe a platform implemented in VHDL which offers a high degree of parametrization [46]. This effort highlights an important groundwork for more recent FPGA deep learning achievements, and marks a shift in focus to CNNs for more application-oriented designs.

3.2 Convolutional Neural Networks on FPGAs

Similar to the MLP, the first FPGA implementations of convolutional neural networks (CNN) began appearing in the mid-to-late 90's. Cloutier et al. were among the first to explore these efforts, but were strongly limited by FPGA size constraints at the time, leading to the use of low-precision arithmetic [47]. In addition, because FPGAs at this time did not contain the MAC units that are present in today's FPGAs, arithmetic was also very slow in addition to being resource expensive. Since this time, FPGA technology has changed significantly. Most notably, there has been a large increase in the density of FPGA fabric, motivated by the decreasing feature (transistor) size, as well as an increase in the number of hardened computational units present in FPGAs. State-of-the-art FPGA implementations of CNNs take advantage of both of these design improvements.

To the best of our knowledge, state-of-the-art performance for forward propagation of CNNs on FPGAs was achieved by a team at Microsoft. Ovtcharov et al. have reported a throughput of 134 images/second on the ImageNet 1K dataset [6], which amounts to roughly 3x the throughput of the next closest competitor, while operating at 25 W on a Stratix V D5 [48]. This performance is projected to increase by using top-of-the-line FPGAs, with an estimated throughput of roughly 233 images/second while consuming roughly the same power on an Arria 10 GX1150. This is compared to high-performing GPU implementations (Caffe + cuDNN), which achieve 500-824 images/second, while consuming 235 W. Interestingly, this was achieved using Microsoft-designed FPGA boards and servers, an experimental project which integrates FPGAs into datacenter applications. This project has claimed to improve large-scale search engine performance by a factor of two, showing promise for this type of FPGA application [49].

Other strong efforts include the design proposed by Zhang et al., referenced above as the closest competitor achieving a throughput of 46 images/second on a Virtex 7 485T, with an unreported power consumption [50]. In this paper, Zhang et al. show their work to outperform most of the main strong competitors in this field, including [51, 52, 53, 54, 55]. Most of these implementations contain architecturally similar designs, commonly using off-chip memory access, configurable software layers, buffered input and output, and many parallel processing elements implemented on FPGA fabric (commonly used to perform convolution). However, important FPGA specific differences exist, such as using different memory sub-systems, data

transfer mechanisms, soft-cores, LUT types, operation frequencies, and entirely different FPGAs, to name a few. As a result, it is hard to determine specific optimal architecture decisions, as more research is needed.

Since pre-trained CNNs are algorithmically simple and computationally efficient, most FPGA efforts have involved accelerating the forward propagation of these models, and reporting on the achieved throughput. This is often of most importance to application engineers who wish to use pre-trained networks to process large amounts of data as quickly and efficiently as possible. However, this only represents one aspect of CNN design considerations for FPGAs, as accelerating backward propagation on FPGAs is also an area of interest. Paul et al. were the first to completely parallelize the learning phase of an MLP on a Virtex E FPGA in 2006 [39]. Other notable efforts include Savich et al. who demonstrated a scalable MLP back propagation implementation on FPGAs that achieved orders of magnitude greater performance over software implementations [56].

3.3 Summary

In this chapter, we reviewed contemporary literature related to the implementation of MLPs and CNNs on FPGAs. We reviewed this literature in a chronological order where possible, giving a general impression of the history of MLPs and CNNs on FPGAs. We notice that, to the best of our knowledge, no other work has focused on the practical integration of FPGAs into common deep learning frameworks.

Chapter 4

Methodology

While the conventional approach of using GPUs has been a valuable contributor to the success of deep learning, hardware acceleration is becoming even more important as the scaling and complexity of these techniques continues to grow. As a result, there is an increasing research interest in alternative acceleration schemes and architectures, and in this work we explore FPGAs as an alternative architecture for deep learning acceleration.

This chapter discusses the methodology of this work as it relates to the development of a practical FPGA acceleration platform for deep learning. We begin by discussing the framework from a high level, and make important comparisons to the conventional GPU-based approach from a parallel programming perspective. Next, we discuss the practicality of this framework, in relation to several important challenges. Finally, we discuss the important details of this framework, including memory structure, device kernel design strategies, and optimizations which are introduced at the layer, device kernel, and bit level.

4.1 Deep Learning Acceleration using FPGAs

To best understand the methods employed in this work, let us revisit our motivating question:

Can FPGAs be used as a practical acceleration platform for deep learning?

This objective can be broken down into a few fundamental parts, which can be addressed and discussed independently. At the heart of this objective is the investigation of an *FPGA platform for deep learning*. While many platforms for deep learning exist, as were discussed in Chapter 2 (e.g. Caffe, Theano), these platforms favour GPU backends. In this work, we focus on the development of a platform for deep learning on FPGAs which functions similarly to the conventional GPU-based approach. This development includes both the theoretical discussion of a design flow which accommodates FPGAs, and the actual implementation of the framework. This topic is introduced in Section 4.1.1, and discussed throughout Section 4.2.

Next, the *acceleration* component of this work involves the investigation of optimization techniques for deep learning that are specific to FPGAs. While this work focuses on the high level ideas of deep learning on FPGAs and is not solely focused on performance, there are many design choices made which are performance driven. As such, we focus our work on developing model-specific optimizations which accelerate CNNs, and provide an empirical assessment of their performance. This topic is introduced in Section 4.1.2 with important details outlined in Section 4.2, and the results are discussed in Chapter 5.

Finally, the concept of *practicality* is essential to this work. While contemporary work has been successful in demonstrating the performance of FPGAs for deep learning tasks, much of this work does not address the practical challenges of integrating FPGAs into software frameworks. As a result, it is unclear if software-oriented deep learning practitioners can benefit from the use of FPGAs. The discussion of practicality in this work is centered around a few important challenges posed by using FPGAs, such as design effort, design iteration, model exploration, and multi-device parallelism. This topic is introduced in Section 4.1.3, and is continuously touched on with the framework details of Section 4.2.

4.1.1 FPGA-Based Deep Learning Frameworks

As discussed in Chapter 2 and summarized in Table 2.1, most popular deep learning frameworks have limited or incomplete support for OpenCL. As well, given the non-standard OpenCL support for FPGAs, frameworks with standard OpenCL backends will not work with FPGAs out-of-the-box, and would need significant structural changes to make this accommodation. Given these difficulties, there are currently no widely-used deep learning frameworks which support FPGAs for acceleration. In response, the main deliverable of this thesis is the development of a deep learning framework which includes support for FPGAs, and addresses important limitations of similar work.

From the perspective of the deep learning practitioner (end user), the user experience for a deep learning framework with FPGA support remains the same as the GPU alternative. In fact, the goal of a well-constructed framework should be to keep the user agnostic to the hardware implementation details, and simply give the choice of platform for acceleration at runtime. The abstraction hierarchy is therefore very similar to the GPU-based approach, as seen in Figure 4.1, with the differences emphasized with asterisks.

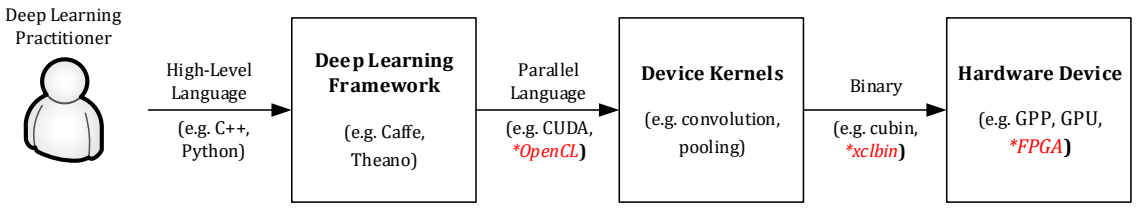


Figure 4.1: In the FPGA-based deep learning framework abstraction hierarchy, the differences (*) from the GPU-based approach are not visible to deep learning practitioners.

By using a parallel programming language which supports FPGAs, we are able to design device kernels which compile into binary (e.g. `xclbin`) which can be used to program an FPGA. It should also be noted that, since these changes are far removed from the user (i.e. to the right in Figure 4.1) these differences are not visible to the typical deep learning practitioner. However, these changes are important to anyone designing at a lower level, such as directly designing device kernels. While the framework developed in this work uses this model of abstraction hierarchy, there are several challenges unique to FPGAs which make

this development more difficult than the GPU-based alternative, discussed in more detail in section 4.1.3.

4.1.2 FPGA Parallelism

While GPPs and GPUs rely on executing parallel threads on multiple cores, FPGAs have the ability to create customized hardware circuits, allowing a more sophisticated type of parallelism called deep pipeline parallelism. Simply put, FPGAs can break down a complicated algorithm into stages, executing different stages on different parts of the data concurrently. Often this helps achieve higher throughput than the equivalent parallel techniques provided by GPUs and GPPs.

Pipelining has particular importance for deep learning. At the layer level, pipelining can be used to execute different layers concurrently to achieve high throughput. Consider a simple example of a CNN with five layers - convolution ($Conv$), rectified linear unit activation ($ReLU$), local response normalization ($Norm$), max pooling ($Pool$), and a fully connected ($Full$) layer. Assuming that each layer has an equivalent execution time of one time unit, Figure 4.2 shows the throughput over 10 time units for several parallelization schemes, where each group of coloured blocks represents a single mini-batch of data that is being processed. This example is inspired by a whitepaper from Acceleware [57].

For the single input single data (SISD) scheme, execution is performed as a single thread, and so 2 mini-batches of data are processed over 10 units of time. The throughput for SISD, which is representative of how a single core GPP would handle this problem, is 1 work-item per 10 clock cycles. For the single input multiple data scheme, assuming 3 work-items can be processed in parallel, a total of 6 mini-batches of data are processed. The throughput for the SIMD parallelism, which is found in GPUs, is 3 work-items per 10 clock cycles. For the pipeline parallelism scheme, a total of 6 mini-batches of data are processed, which is equivalent to the SIMD approach. However, the pipeline parallelism strategy, which is possible on FPGAs, has a max throughput of 5 work-items per 10 clock cycles, which is higher than both the SIMD and SISD approach. Though SIMD and pipeline parallelism processed the same amount of data in this example, since the pipeline parallelism experiences a higher throughput it would become much more efficient as additional mini-batches are processed.

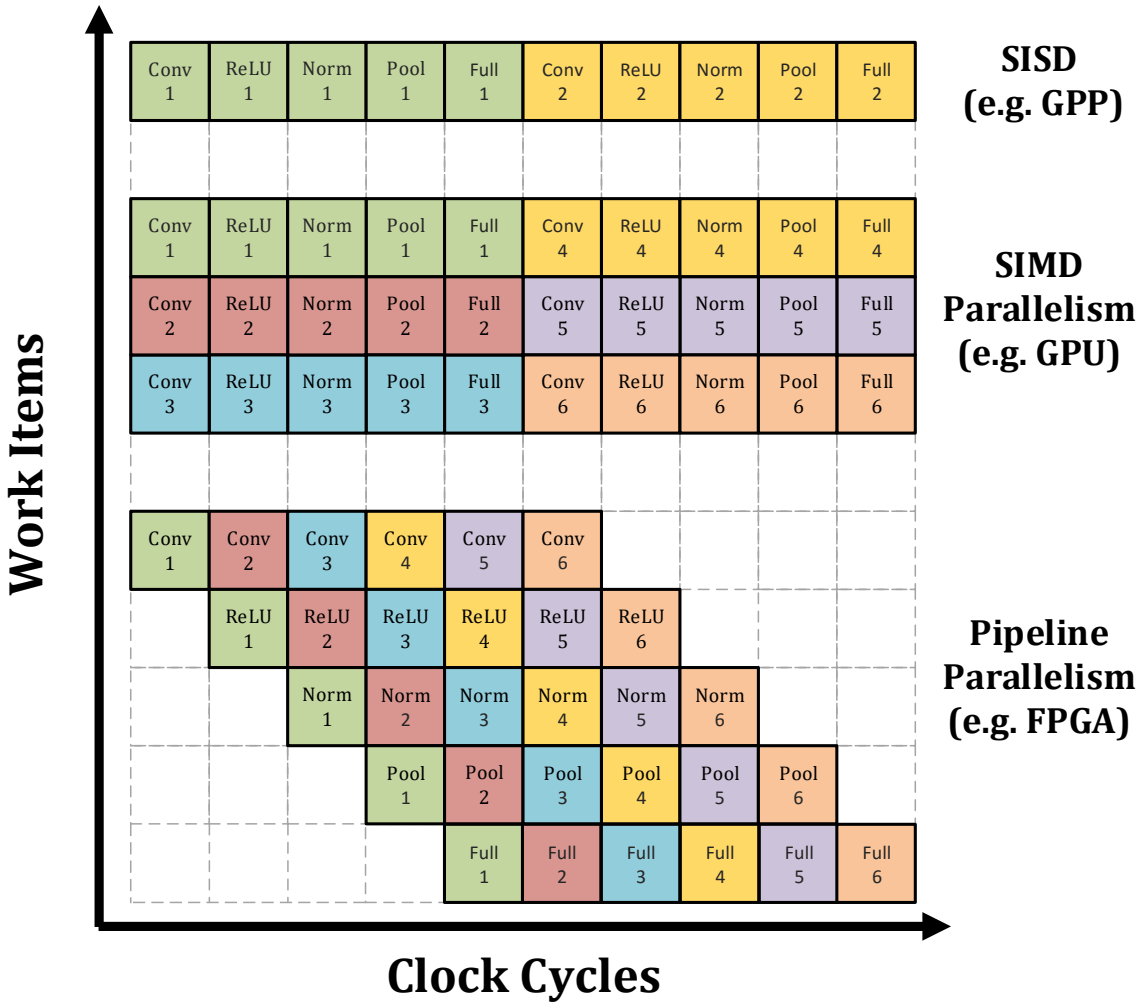


Figure 4.2: Consider a simple 5-layer CNN. Let each block represent one layer and each group of coloured blocks represent one "mini-batch" of data. In 10 units of time, both the FPGA and GPU complete 6 "mini-batches" of data compared to 2 on the GPP. The FPGA, using pipeline parallelism, achieves the highest maximum throughput of 5 work-items.

Figure 4.3 shows a visualization of how this example is implemented on an FPGA to achieve pipeline parallelism. Notice that, while the computational graph is identical to that of the GPU-based approach, the FPGA is able to execute multiple layers concurrently, while performing fewer memory transactions

with the host. Kernel to kernel communication is accomplished using FIFOs local to the FPGA. Another interesting feature is the ability for FPGAs to replicate circuits in hardware to achieve another level of parallelism. To use SDAccel terminology, we can instantiate multiple compute units, where each compute unit contains an identical copy of the circuit, including its own local memory. To relate to the previous example, each compute unit containing a full copy of the network could each handle one separate image concurrently. In practice, one must explore the tradeoff of exhausting parallelism within a circuit compared to across replications of that circuit. Further discussion of this parallelism strategy, as well as best practices, is found in section 4.2.4 discussing pipeline layers in the FPGA Caffe framework.

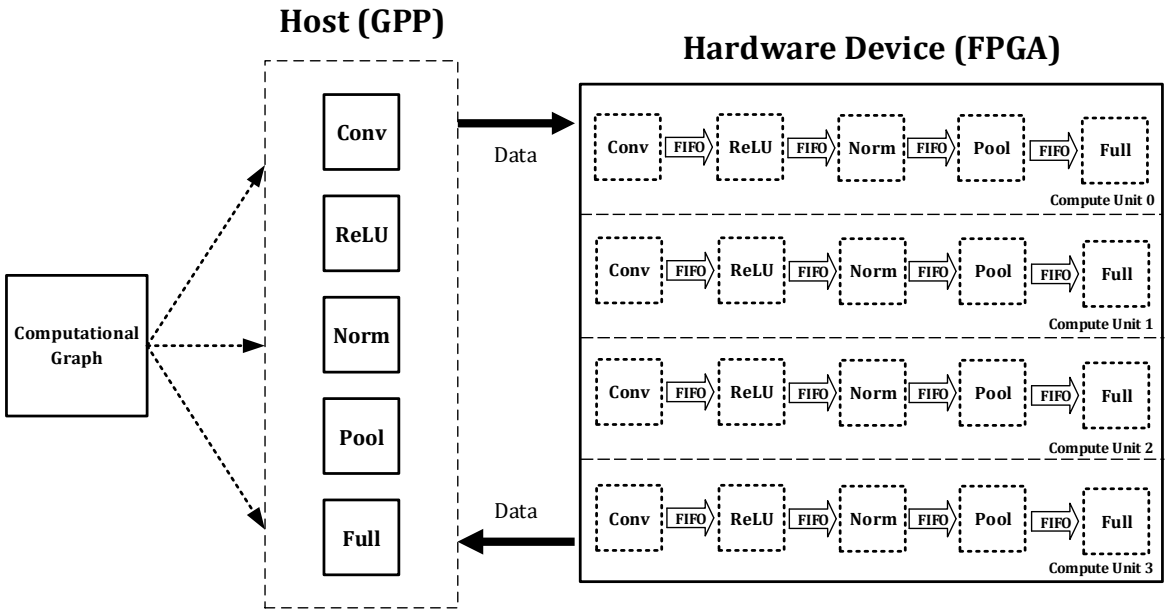


Figure 4.3: For FPGA acceleration, computational graphs are broken down into modules (e.g. layers of a CNN), which can be grouped together in a single binary. To maximize throughput, circuits can also be replicated (compute units in SDAccel). In contrast to GPUs, several unique modules can be executing at any given time and data can be exchanged between modules on the FPGA using local data structures (i.e. FIFOs), alleviating the need for each module to exchange data with the host.

Pipeline parallelism is commonly applied at the loop level, where instructions can be pipelined across loop iterations to keep resources busy. On GPUs, the most efficient SIMD parallelism is achieved when loops carry no dependencies across loop iterations. It is often difficult to avoid loop-carried dependencies, and so GPU resources may remain idle leading to suboptimal performance. However, FPGAs are able to achieve

more performance by executing the next iteration of the loop concurrently as soon as the dependency is fulfilled. This is called loop pipeline parallelism, and is typically accessed through vendor extensions of the OpenCL framework, such as the command `xcl_pipeline_loop`.

4.1.3 FPGA Challenges

Recent work on using FPGAs for deep learning has shown promising results for metrics such as throughput and performance per watt [10]. However, much of this work does not focus on the practicality of FPGAs compared to GPUs for typical deep learning workflows. Some of the commonly ignored challenges specific to FPGAs are listed below.

1. **Design Effort:** Many current solutions for deep learning on FPGAs involve a large amount of hardware design effort using FPGA-specific languages at a high-level such as HLS, or a low-level such as HDL or RTL. These languages require hardware-specific knowledge to use effectively, and are not accessible to deep learning scientists who are accustomed to software designs. In addition, these languages do not integrate well with deep learning frameworks, and so this work is often not useful for practical deep learning. For current GPU-based solutions, the CUDA framework provides an accessible software interface to the underlying hardware.

   To address this issue, this work uses OpenCL as an interface between the deep learning software and FPGA hardware. As a high-level parallel programming language, this interface is much more familiar to software programmers who are accustomed to the CUDA interface. In this work, both the host and device kernel code are written using OpenCL, meaning much of the convenience of using deep learning frameworks is retained.

2. **Design Iteration:** The synthesis time for complex FPGA designs can be very large (tens of minutes to hours), making it unsuitable for just-in-time compilation methods employed in some deep learning frameworks. For current GPU-based solutions, CUDA code has a very small compile time (milliseconds) which makes just-in-time compilation practical.

   To address this issue, this work employs an offline compilation strategy, where device binaries are

created in a one-time offline compilation. At runtime these binaries are used to program the FPGA with a very small latency (tens of milliseconds), allowing a similar workflow experience to that of using a GPU with just-in-time compilation. Since certain hyper-parameter settings are fixed within each binary (e.g. convolution filter size), several binaries are compiled for each popular hyper-parameter setting. We argue that, since input and layer sizes are still variable in these binaries, and only a few hyper-parameter combinations are commonly used (e.g. convolution filter size of 3×3, 5×5, 7×7), this design choice is practical.

3. **Model Exploration:** Most current work focuses only on deployment on FPGAs, and does not address training. Deployment is a much simpler problem at the hardware level as model parameters are fixed, but training is also an important problem in acceleration as models grow in size and complexity. For current GPU-based solutions, frameworks that implement both training and deployment techniques are used in practice.

   While this work focuses only on deployment, both training and deployment are easily supportable, since this work extends the Caffe architecture. Currently, we have implemented the deployment (forward) kernels necessary for common CNNs architectures such as Alexnet [6] and will implement training (backward) support in the future. Given the modularity between the solver, network, and layer in Caffe, only the layers need to be modified to accommodate backward propagation. Each layer class has a forward and backward method for supporting multiple hardware devices (e.g. `forward_cpu`, `forward_gpu`). For example, when executing training on an FPGA, the solver (e.g. stochastic gradient descent, RMSprop) calls the `forward_fpga` method for each layer of the network to produce the output and loss, then calls the `backward_fpga` methods of each layer to generate gradients, and produce a weight update which minimizes the loss. Given the implementation of the `forward_fpga` methods, supporting training involves implementing the `backward_fpga` methods, with the infrastructure needed for defining the network and performing weight updates already in place.

4. **Multi-Device Parallelism:** With the scaling of deep learning continuing to grow, multi-device acceleration schemes are becoming essential. The issue of limited memory on GPUs to support model parallelism is addressed by scaling to using multiple-GPUs. Even with simple data parallelism, it is often desirable to scale up to an amount of GPUs larger than what would traditionally fit in a single node (e.g. 4 or 8 GPUs). While multi-GPU in a single node allows high performance, technologies

such as NVIDIA GPUDirect allows high performance across nodes by allowing GPU-GPU memory transfers between GPUs on different nodes without host memory intervention. Largely, the challenges of multi-GPU support are algorithmic given the standardization of networking infrastructure. For current GPU solutions, the use of multi-GPU clusters with Infiniband interconnects and MPI is a common approach [1]. However, there is no consensus technology enabling multi-FPGA systems that is available to deep learning scientists.

This work does not address the issue of multi-FPGA parallelism, but acknowledges the importance of this functionality moving forward. On the FPGA side, the problem is mainly in hardware, as interconnects and multi-FPGA technologies are non-standard. Once this hardware is standardized, FPGAs can exploit the parallel algorithms that have been developed on GPPs and GPUs.

5. **Economic Factors:** While both FPGA and GPU technologies have a range of devices which appeal to the trade-off of performance and cost, it is typically the case that FPGAs tend to be more expensive than GPUs for a given computational capability [58]. This makes the justification of FPGAs for cost-sensitive applications difficult, as the initial investment for FPGAs can be greater than for GPUs.

   This work addresses the economic factors of FPGA practicality through the assessment of performance metrics which include a measure of power consumption. This assessment is most important for data center applications, where the ongoing electricity costs affected by power consumption greatly outweigh the initial investment for long term economic decisions.

4.2 FPGA Caffe Framework

We demonstrate a practical deep learning framework which addresses the challenges of integrating FPGAs into a deep learning design flow. This framework, FPGA Caffe, is an extension of the popular Caffe deep learning framework. While CUDA provides support for GPPs and GPUs, we extend this support to include FPGAs through OpenCL. Specifically, we use the Xilinx SDAccel v2015.1 [27] tool to facilitate this support.

4.2.1 Deployment Architecture

The structure of support for hardware devices is mirrored after the existing Caffe support for GPP and GPU execution. In addition to the CUDA backend, an OpenCL backend is used to support FPGA execution. The OpenCL backend includes both host code for handling memory synchronization between the host and FPGA, and device code which implements the required deep learning operations on the FPGA. For each layer type used in AlexNet, an OpenCL device kernel implementation was created which accelerates that layer on the FPGA. This is done through implementing `forward_fpga` forward propagation functions in addition to the existing `forward_cpu` and `forward_gpu` functions. As such, there exist three sets of implementations for each layer type in AlexNet: a C++ implementation for GPP execution, a CUDA implementation for GPU execution, and an OpenCL implementation for FPGA execution. At runtime, the user can specify the desired hardware execution device. For example, assuming the user is using the command line interface, the GPU is specified using `-gpu=1`, and the FPGA is specified using `-ocl=1`, where the numeric value indexes the hardware device in the case of multiple devices on the same system. If no flag is specified, the GPP is used by default. Figure 4.4 shows an overview of this architecture for an example where a user wishes to classify an image using an FPGA to accelerate deployment.

4.2.2 Memory Architecture

In Caffe, data is stored in a wrapper called a *blob*, which is a unified memory interface abstracting away the synchronization details between the host and hardware device. Blobs are used to store batches of images, parameters, and gradients, and pass them between layers. While a blob is an N-dimensional array stored in a C-contiguous fashion, blobs are typically 4-dimensional (number of images $\times$ number of channels $\times$ height $\times$ width) and row-major. While this interpretation is specific to image data, blobs in Caffe can be reinterpreted to represent non-image data. For each layer in a Caffe model, there is a bottom blob, which represents data being passed from the previous layer, and a top blob, which represents data being passed to the next layer. For the Caffe GPU backend, to facilitate synchronization each blob uses a synchronized memory class `SyncedMem` which has both mutable and static access methods, depending on whether or not one wishes to change the values of the blob. If the blob is located on the GPP, and the blob is requested

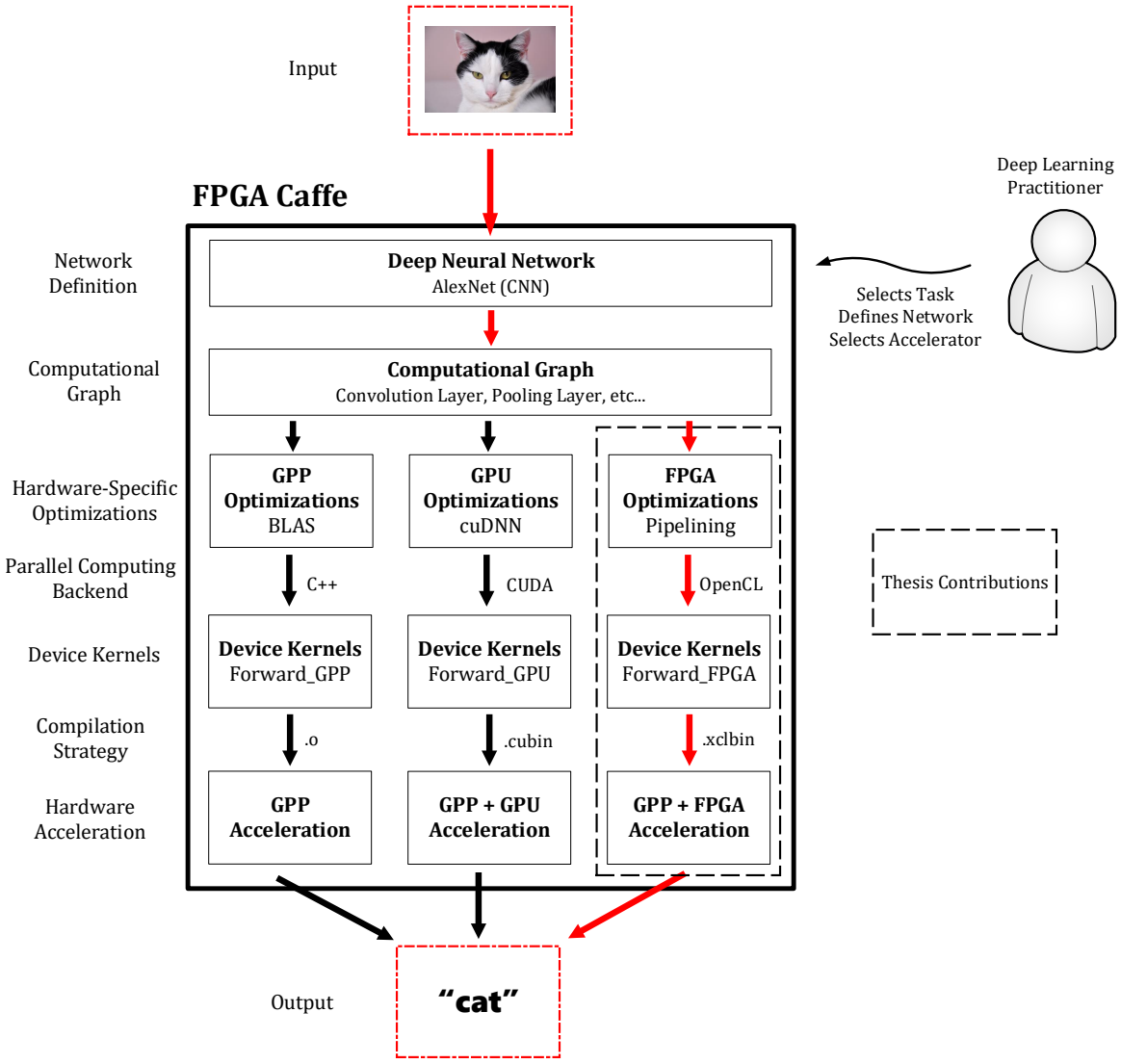


Figure 4.4: This figure demonstrates a more complete picture of the FPGA Caffe framework compared to Figure 1.1 in Chapter 1. We see that each hardware-specific data path shares the same computational graph, which is composed of layers specific to the AlexNet CNN. Hardware-specific optimizations for FPGAs are achieved through strategies such as pipelining, while for GPPs this can be achieved through different linear algebra (BLAS) packages, and for GPUs this can be achieved through optimized deep learning libraries (cuDNN). It should be noted that FPGA the device kernel binaries (.xclbin) are compiled offline, and are used to program the FPGA at runtime.

for a GPU operation, the SyncedMem class will use CUDA API calls to copy the data from GPP to GPU.

For the FPGA Caffe implementation, memory synchronization is achieved in a similar fashion. Instead of CUDA API calls for data transfers, OpenCL API calls are used to regulate memory transfers between the host and FPGA. Specifically, the location of a blob is tracked using a pointer, which can either be in the state called `HEAD_AT_OCL` indicating that it is residing on the FPGA, or `SYNCED` meaning that the data has been synchronized with the host and is located on the GPP. For example, if calling the method `to_cpu` (indicating a desired memory transfer of FPGA to host GPP) host memory is allocated and the OpenCL API function `clEnqueueReadBuffer` will be executed, which reads memory from the FPGA to the host GPP.

4.2.3 Device Kernel Design

By using OpenCL for design, software programmers are able to create FPGA kernels through our framework with little or no hardware knowledge required. In this work, we restrict ourself to optimizations that are achieved through SDAccel vendor extensions, such as loop unrolling and loop pipelining. To demonstrate the design effort of these vendor extensions, consider an example where the summation of all pixel values in a image is required. When using OpenCL to design this kernel for a GPU, a typical design would use tiling and kernel concurrency. Since the work-item is indexed using `get_global_id(0)`, and the size of the work-group is given using `get_global_size(0)`, we could express the loop as follows:

```
for (int i = get_global_id(0); i < IMG_SIZE; i += get_global_size(0)) {
        SUM += IMG[i];
}
```

For regular image sizes, where `IMG_SIZE` = `get_global_size(0)`, the compiler will create as many work-items as needed to index the entire image, and if the number of GPU cores available is greater than or equal to the image size, these work-items will execute concurrently. If the image size is irregular and

`IMG_SIZE` != `get_global_size(0)`, the compiler will tile kernels over the problem space as necessary.

The approach for FPGAs is very simple. By simply using the SDAccel vendor extension `__attribute__((opencl_unroll_hint))` for unrolling a loop fully, the compiler will execute each iteration of the loop concurrently on the FPGA:

```
__attribute__((opencl_unroll_hint))
for (int i = 0; i < IMG_SIZE; i += 1) {
        SUM += IMG[i]
}
```

For GPUs the kernel is designed from the perspective of the work-item, while for FPGAs the kernel is designed from the perspective of the entire problem space. The device kernel designs used in this work, described below, make heavy use of these vendor extensions for optimization.

Fully Connected Layers

Since fully connected layers can be expressed as a simple general matrix to matrix multiplication (GEMM), the device kernel design for this layer is fairly straightforward, and is usually done in a single nested loop. However, this logic becomes slightly complicated when using vectorization data types available in OpenCL which improve performance. Vector data types are data types which create a wide computation data path by allowing each element in a buffer to represent a vector, as opposed to a scalar which is typically represented by a single buffer element. For example, a `float8` data type handles a vector of 8 floating point values at each element of the buffer, where each of the 8 elements can be accessed as: `buffer.s0`, `buffer.s1` ... `buffer.s7`. This vector data type greatly improves performance over unvectorized code, as each element of each vector can be accessed in the same clock cycle. For a given loop which requires accessing each element of a buffer, this would amount in 8× fewer loop iterations. However, this complicates our logic for computing fully connected layers, as we need a separate step to perform summations on each vector element. Algorithm 1 shows pseudocode for the fully connected OpenCL device kernel, where the

GEMM is broken down into two seperate loops, one for calculating the product, and one for calculating the sum. It should be noted that each of these loops have an SDAccel directive `xcl_pipeline_loop`, which tells the SDAccel compiler to look to optimize these loops by both full unrolling and pipelining instructions within iterations. This algorithm uses `float8` vector data types, meaning loop bounds are reduced by a factor of 8.

Algorithm 1 Fully Connected OpenCL Device Kernel Pseudocode

```
# define K as input layer size, N as output layer size
function FC(float8* A (in), float8* B (weights), float8* OUT (out))
    __local float8 product[N*K/8]
    __local float8 sum[N*K/8]
    // Copy A from global memory to local memory
    // Copy B from global memory to local memory
    xcl_pipeline_loop:
    for i=0 to i= N*K/8 do
        prod[i] = A[i]*B[i]
    end for
    xcl_pipeline_loop:
    for i=0 to i= N*K/8 do
        sum[i] += product[i].s0 + ... + product[i].s7
    end for
    // Copy results from local memory (sum[N*K]) to global memory (OUT)
end function
```

In addition to vector data types, the other interesting things to note about this kernel is that buffers which are required as function parameters (namely input, weights, and output data) are copied to local memory when being accessed. This is crucially important, as if these buffers are not copied to local memory, they are copied from global memory each time they are required. This would cause serious performance limitations, and global memory accesses are much slower than local memory.

Convolutional Layers

In terms of computational requirements, convolution is the most expensive layer in CNNs. Algorithmically, convolution involves looping over both the input image and convolution filter, performing multiplications and additions at each step. While the convolution implementation used in this work is far from optimal, it demonstrates a viable solution that tailors optimizations to the FPGA device. Similar to fully connected layers, the input, convolution filters (weights), and output are all copied from global to local memory while being accessed inside of this kernel. One interesting design choice to notice is that the loop pipelining directive and a local memory copy are applied not at the outermost loop, but towards the inside of the 4-dimensional nested loop. This is largely due to the fact that the loop dimensions grow rapidly given the size of the images and filters, and so pipelining and unrolling all iterations of the loops makes synthesis of the circuit impossible. Given how SDAccel pipelined loops cannot also be partially unrolled, this design choice was largely a limitation of the current state of the tools and hardware. While we cannot determine exact speed-up of unrolling the entire loop due to these synthesis issues, we can assume that the speed-up is proportional to the number of sequential loop iterations that are unrolled, given our experiences with full unrolling of loops for other layers. The convolution pseudocode is found in Algorithm 2.

ReLU Activation

ReLU activations are a very straightforward computation, requiring only a condition statement which retains the value if greater than 0, or outputs a 0 otherwise. This logic is only slightly complicated by the vector data types used in OpenCL to improve performance, as seen in Algorithm 3. It should be noted that since we cannot compute the conditional on all vector elements at the same time, they need to be split apart within our pipelined loop.

Algorithm 2 Convolution OpenCL Device Kernel Pseudocode

```
# define IN_ROWS as number of rows in input, IN_COLS as number of columns
# define FILT_ROWS as number of rows in filters, FILT_COLS as number of columns
# define OUT_ROWS as number of rows in output, OUT_COLS as number of columns
# define PAD as padding, STRIDE as stride
function CONV(FLOAT* A (INPUT), FLOAT* B (FILTERS), FLOAT* OUT (OUTPUT))
    __local int idx_out, idx_in, idx_filt
    // Copy B from global memory to local memory
    OUT_COLS = ((IN_COLS-FIL_COLS+(2*PAD))/STRIDE))+1
    OUT_ROWS = ((IN_ROWS-FIL_ROWS+(2*PAD))/STRIDE))+1
    for i=0 to i= OUT_ROWS do
        for j=0 to j= OUT_COLS do
            // Copy A from global memory to local memory
            xcl_pipeline_loop:
            for k=0 to k= FILT_ROWS do
                for l=0 to l= FILT_COLS do
                    idx_y = i * STRIDE - PAD + k
                    idx_y = j * STRIDE - PAD + l
                    if 0 ≤ idx_y ≤ IN_ROWS and 0 ≤ idx_x ≤ IN_COLS then
                        idx_out = i * OUT_COLS + j
                        idx_in = in_y * IN_COLS + in_x
                        idx_filt = k * FILT_COLS + k
                        OUT[idx_out] += A[idx_in] * B[idx_filt]
                    end if
                end for
            end for
        end for
    end for
    // Copy OUT from local memory to global memory
end function
```

Algorithm 3 ReLU Activation OpenCL Device Kernel Pseudocode

```
# define N as output size
function ReLU(float8* A (input), float8* OUT (output))
    // Copy A from global memory to local memory
    xcl_pipeline_loop:
    for i=0 to i= N*/8 do
        OUT[i].s0 = (A[i].s0 < 0) ? 0: A[i].s0)
        ...
        OUT[i].s7 = (A[i].s7 < 0) ? 0: A[i].s7)
    end for
    // Copy OUT from local memory to global memory
end function
```

Pooling Layers

The max pooling device kernel in this work uses similar techniques to other kernels, using local memory copies and pipelined loops. It should be noted that, since there exists some logic inside loops that are not the innermost loop, this nested loop is considered imperfect and is not ideal for optimization. We use the definition of perfect loop as defined in the Xilinx HLS tool, where only the innermost loop has contents, there is no logic between loop statements, and loop bounds are constant. Further improvements to Algorithm 4 would come from finding a way to ensure the entire nested loop is perfect.

Local Response Normalization Layers

The implementation of local response normalization layers is similar to previous examples, with local memory copies and pipelined loops being used. We can conclude, from the pseudocode seen in Algorithm 5, that since each mathematical expression relies on the previous, the compiler will be unable to fully pipeline many of these expressions.

Algorithm 4 Max Pooling OpenCL Device Kernel Pseudocode

```
# define IN_ROWS as number of rows in input,IN_COLS as number of columns
# define OUT_ROWS as number of output rows, OUT_COLS as number of columns
# define POOL_ROWS as number of filter rows, POOL_COLS as number of columns
function POOL(FLOAT8 A (INPUT), FLOAT8 OUT (OUTPUT))
    __local int value, maximum
    // Copy A from global memory to local memory
    xcl_pipeline_loop:
    for i=0 to i= OUT_ROWS do
        for j=0 to j= OUT_COLS do
            in_x = i * STRIDE
            in_y = j * STRIDE
            for k=0 to k= POOL_ROWS do
                for l=0 to l= POOL_COLS do
                    value = A[(in_y + k) * IN_COLS + in_x + l]
                    if value < maximum then
                        maximum = value
                    end if
                end for
            end for
            OUT[i * OUT_COLS + j] = maximum
        end for
    end for
    // Copy OUT from local memory to global memory
end function
```

Algorithm 5 LRN OpenCL Device Kernel Pseudocode

```
# define IN_ROW as number of input rows,IN_COL as number of columns
# define LOCAL_SIZE as the number of neighbouring images to normalize
# define ALPHA, BETA, INV_SIZE as LRN parameters
function LRN(FLOAT* A (INPUT), FLOAT* OUT (OUTPUT))
    // Copy A from global memory to local memory
    xcl_pipeline_loop:
    for i=0 to i=IN_ROW do
        for j=0 to j=IN_COL do
            for k=0 to k=LOCAL_SIZE do
                value = A[(k * IN_ROW + i) * IN_COL + j]
                scale = value * value
            end for
            scale = scale * ALPHA * INV_SIZE + 1
            arg = -1 * BETA * log(scale)
            OUT[i * OUT_COL + j] = A[(IN_ROW + i) * IN_COL + j] * exp(arg)
        end for
    end for
    // Copy OUT from local memory to global memory
end function
```

4.2.4 Layer Extensions

In addition to improving design effort at the kernel level, the proposed framework attempts to make optimizations easier at the layer level. To realize this, we introduce layer extensions as a part of the FPGA Caffe framework. To demonstrate why these are used, consider the standard approach, where GPU kernels are exchanged for FPGA kernels. Where each GPU kernel takes fractions of a millisecond to compile and execute at runtime, programming the FPGA over PCIe can take anywhere from 150 - 300 ms for kernels of small to medium complexity. This is a significant bottleneck in performance, as for small kernels the time to program the FPGA can be as much as 20x larger than the time to complete the computation. To deal with this bottleneck, we introduce two layer-based extensions to the Caffe framework; programming layers and pipeline layers.

Programming layers are introduced as a method to allow explicit control of how the FPGA is programmed at the model level. These layers are created as any other layer in a Caffe model declaration, but with only the binary and kernel names as arguments. This layer simply passes through the data untouched, programs the FPGA with the indicated binary over PCIe, and creates pointers in memory to the kernels now residing on the FPGA. For benchmarking, the timing of this layer indicates how long it took to program the FPGA.

Pipeline layers are introduced as a method for reducing the amount of times the FPGA is programmed, reducing the number of global memory reads, and increasing throughput. The primary motivation is to reduce the number of programming layers by packaging multiple kernels into a single binary container. Figure 4.5 demonstrates a simple example with one convolution and one pooling layer.

In the standard GPU-based approach, the FPGA is programmed once before the convolution layer *Conv*, and once before the pooling layer *Pool*. In addition, global memory operations are performed before and after the execution of each kernel. The logical improvement is the single kernel non-pipeline approach, where both convolution and pooling are combined sequentially into a single kernel inside of a single binary container. This reduces both the amount of required programming layers and global memory operations by half, as the data shared between each operation stays in private memory on the FPGA. However,

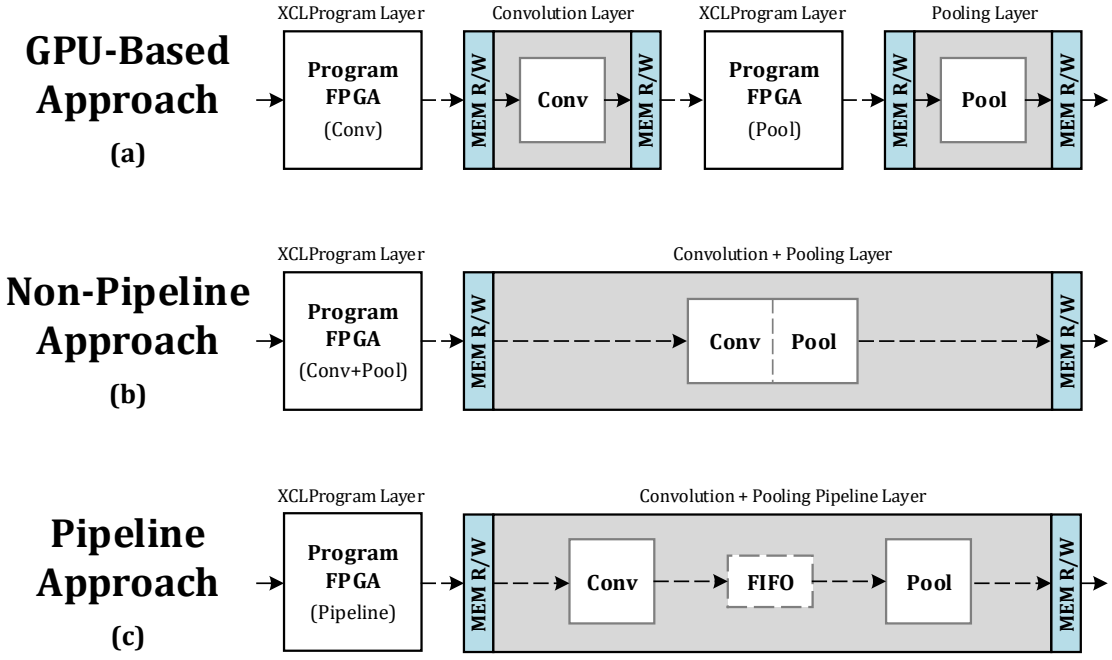


Figure 4.5: Consider a simple network using one convolution and one pooling layer. (a) The GPU-based approach requires 2 program layers and 4 global memory operations (MEM R/W). (b) Using a single kernel, the non-pipeline approach requires only 1 program layer and 2 global memory operations, but the kernels cannot execute concurrently. (c) The pipeline approach is similar but kernels can execute concurrently and exchange data using a FIFO.

since operations are organized sequentially in a single kernel, there is no ability to execute each operation concurrently. This can be further improved with a pipeline approach, where each kernel exists separately within the same binary container, but can execute concurrently. In this approach, data is exchanged between kernels using a FIFO in private memory. Experimental results for the pipeline layers compared to other approaches are introduced in Chapter 5.

4.2.5 Design Iteration

A key contributor to fast and efficient design iteration is the ability to quickly design, compile, and debug kernels. When designing for GPUs, kernels are compiled into instructions, meaning compile time is short. However, when designing for FPGAs kernels are synthesized into hardware circuits, which is a much more computationally-intensive task. To deal with this, our framework uses a staged offline compilation strategy to more closely mimic the compilation strategy used for GPUs and GPPs. By "compiling" kernels offline, the binary can then be used to program the FPGA at runtime. The stages are important for kernel development, as the hardware does not have to be fully synthesized to perform function or application debugging. The stages used for compilation are software emulation (for functional debugging), hardware emulation (for application debugging), and hardware synthesis (for performance debugging). The compilation times for typical kernels used in the FPGA Caffe framework are presented in Table 4.1.

Table 4.1: Experimental results for FPGA kernel compilation times.

Stage	Purpose	Device	Time (s)
Software Emulation	Functional debugging	CPU Host	5-8
Hardware Emulation	Application debugging	Virtual FPGA	25-50
Hardware Synthesis	Performance debugging	FPGA	1800 - 7200

During the software emulation stage, kernel logic is compiled for a GPP host, and both semantic and functional errors can be found quickly as the compilation time is very small. During hardware emulation, kernel logic is synthesized for a virtual FPGA fabric, and detailed reports are given which allow the designer to see efficiency estimations, as well as identify potential performance bottlenecks. A virtual FPGA fabric is equivalent to a hardware simulator which emulates the structure of the FPGA for fast testing. In SDAccel, this allows the functional verification of having multiple circuit replications (compute units). Software emulation alone cannot simulate a design with multiple compute units (it only understands a single compute unit), so hardware emulation is needed to verify functional correctness of designs with multiple compute units. Since the compilation time for hardware emulation is also small, it is easy to iterate through multiple kernel designs very fast. Finally, during the hardware synthesis stage a full circuit is synthesized for the FPGA, and the generated reports give real performance values. Since the compile time for this stage is very slow, it is only advised as a final step [57].

4.3 Summary

In this chapter, we discussed the methodology employed in this thesis, which included an overview of the FPGA Caffe framework structure and functionality. As well, we compare the use of FPGAs in deep learning frameworks directly to the GPU-based approach, by discussing how it affects the deep learning practitioner, how parallelism is achieved, and the challenges of using FPGAs for deep learning. Throughout this chapter, as well as the remainder of the thesis, we commonly refer back to these challenges as we discuss our efforts to to address them.

Chapter 5

Results

In order to validate the functionality of this framework, and justify design choices from a performance perspective, we provide an empirical assessment using common benchmarking techniques. This is accomplished through the integration of benchmarking functionality within the framework itself, which allows software consistency when comparing across different hardware. Performance is evaluated across execution time, throughput, and power consumption using synthetic data, and is restricted to the AlexNet CNN architecture [6]. Additionally, profiling information such as FPGA resource usage is provided for consideration and discussion. We organize our results first by layer in Section 5.2, by model in Section 5.3, isolate our pipeline layer results in Section 5.4, and finally we present our power consumption results in Section 5.5. Given the generous amount of profiling information available from these experiments, we attempt to include only the most relevant information in this chapter.

5.1 Experimental Setup

The class of techniques belonging to deep learning is large and complex, and so it is difficult to develop and validate a framework which captures this breadth. We therefore limit our development in this framework to one popular architecture, the CNN, as its success in both industry and academia has made it one of the most widely used deep learning architectures today. Specifically, we choose the AlexNet CNN architecture as a reference for benchmarking. This model is frequently used as a benchmarking reference across many types of high performance computing tasks, as it has high accuracy for classification, contains a high amount of complexity requiring hardware acceleration, and is familiar to many deep learning practitioners. Our investigation looks at each layer of AlexNet individually, as well as the full architecture.

This work compares FPGA performance to that of both a GPP and GPU. The FPGA board used is an Alpha Data ADM-PCIE-7V3 based on the Xilinx Virtex-7 XC7VX690T FPGA running at 200 MHz. It has two independent channels of 8GB DDR3 memory, and uses a PCIe x8 form factor. This FPGA was chosen mainly for its support of OpenCL, as very few FPGAs today support OpenCL. The GPP used is a Intel Xeon E5-2637 v2 running at 3.50 GHz with 8 cores. The GPU used is an NVIDIA Quadro K600 GK107GL running at 876 MHz. It has 2 GB of DDR3 memory, and uses a PCIe x16 form factor. While none of these devices are considered state-of-the-art at the time of this work, we believe that they give reasonable performance estimations which allow us to make broad conclusions about each type of architecture. All three of these devices are connected through PCIe slots of the same host workstation. The OpenCL environment used is the Xilinx SDAccel 2015.1 software tool, and for both GPP and GPU execution, CUDA 7.0 was used. For GPP device kernel execution, the ATLAS linear algebra (BLAS) package was used, which restricts multi-threading on our CentOS release 6.8 workstation.

Timing is measured using the Caffe timer benchmarking functions, accessed through the command line. The timer uses the POSIX time standard, meaning that time is recorded using a sub-second resolution clock based off the unix wall clock of the host system. For profiling, the SDAccel runtime profiler was used to collect information. The resource utilization is expressed in terms of compute units (CU), flip-flops (FF), look-up tables (LUT), digital signal processing blocks (DSP), and block RAM (BRAM). Power consumption is measured using an electricity usage meter (Kill A Watt) which claims accuracy to within

0.2 %. This meter is placed between the host computer and wall power source, and power consumption is tracked during the continuous usage of a hardware device for accelerating the Caffe AlexNet model, averaged over time. While hardware-specific methods of measuring power consumption can be used, this method was deemed most appropriate for these experiments as the method remained constant and fair across all hardware devices used. Finally, throughput is reported in terms of floating point operations per second (FLOPS). This value is estimated using careful assessment of the underlying implementation. For example, in the fully connected layer implementation, the main logic loops over each of the input (N) and output (M) units, performing a multiply and sum at each iteration. As a result, since two floating point operations are performed per iteration, we can estimate the FLOP count as $2 * N * M$, and simply divide by the execution time to find the FLOPS. More detailed FLOP calculations are found in Appendix A.

5.1.1 FPGA Hardware

The Alpha Data ADM-PCIE-7V3 was chosen largely for its official support of OpenCL, namely the Xilinx SDAccel development environment which compiles OpenCL programs to execute on a Xilinx FPGA (Virtex-7 XC7VX690T). To enable support for the SDAccel environment, this FPGA contains two distinct regions, the static and programmable region. The static region contains the entire PCIe core, as well as global memory interfaces, and is required to facilitate communication between the host and OpenCL device kernels. The programmable region contains the general programmable fabric, where device kernels are implemented and executed. As a result, while the entire FPGA resources are contained within both the static and programmable region, the device kernels only have access to the programmable region. The resource statistics of these regions are collected in Table 5.1.

Table 5.1: Alpha Data ADM-PCIE-7V3 FPGA Resource Statistics

ADM-PCIE-7V3				
Region	FF	LUT	DSP	BRAM
Static	315,360	157,680	1,224	1,000
Programmable	551,040	275,520	2,376	1,940
Total	866,400	433,200	3,600	2,940

5.2 Layer Benchmarking

We first look at performance in a layer-wise fashion, by implementing several important layers common to CNN architectures, and comparing these implementations running on FPGAs to those created for GPP and GPU execution. The layers chosen are the fundamental layers required to create the full AlexNet architecture. While these layers do not fully encompass all possible CNN architectures, we believe that it offers reasonable complexity and performance which matches many real-world applications. For each layer type, an OpenCL implementation was created and deployed using a Caffe model to facilitate benchmarking. Benchmark timings were recorded using the Caffe `time` command, run on synthetic data and averaged over 10 iterations. For each GPP and GPU implementation, the corresponding C++ and CUDA implementations distributed in the offical Caffe distribution were used. For each layer type, we implement all the corresponding sizes (hyper-parameter configurations) found in the AlexNet model, and report values ± one standard deviation. Resource statistics are found using the SDAccel compiler tool, and presented as a percentage of total resources available in the programmable region.

Fully Connected Layers

Benchmark timings for the fully-connected layer implementations are seen in Table 5.2. Fully-connected layers are parametrized by the input and output sizes, which describe the number of units in the previous and next layer to which full connections exist, known as the "fan-in" and "fan-out", respectively.

Table 5.2: Fully-Connected Layer Benchmark Results

AlexNet Fully-Connected Layers

Layer	Input Size	Output Size	GPP (ms)	GPU (ms)	FPGA (ms)
FC6	9216	4096	47.39 ± 0.3	10.89 ± 0.3	23.43 ± 0.2
FC7	4096	4096	20.27 ± 0.2	4.72 ± 0.2	10.52 ± 0.1
FC8	4096	1000	4.92 ± 0.1	1.19 ± 0.1	2.49 ± 0.1

FPGA resource utilization statistics for each fully-connected layer implementation are seen in Table 5.3. To recall, compute units describe the number of replicated copies of the device kernel which are created on the FPGA. A work-group is mapped to a single compute unit (CU). As such, to calculate the resources required for a single copy of the device kernel, we simply divide the total resource usage by the number of compute units.

Table 5.3: Fully-Connected Layer Resource Utilization

	Fully-Connected Layer								
Layer	CU	FF	FF%	LUT	LUT%	DSP	DSP%	BRAM	BRAM%
FC6	8	58392	10.6	45608	16.6	608	25.6	888	45.8
FC7	8	56736	10.2	42800	15.6	304	12.7	296	15.3
FC8	8	61816	11.2	44648	16.2	320	13.4	232	11.9

From the benchmark timings, we see that for all layers considered, the GPU provides the highest performance in terms of execution time, followed by the FPGA, and finally the GPP. This result is as expected, given the simplicity of the fully-connected layers and the parallel architecture of both the GPU and FPGA. In terms of resource utilization, it is interesting to note that while most resources are used similarly across the three layers, the much bigger $FC6$ required a heavier usage of BRAM to store the additional weights. As well, although a maximum of 10 compute units are allowed by the SDAccel compiler, in practice, only 8 were able to be successfully synthesized.

Convolutional Layers

Benchmark timings for the convolution layer implementations are seen in Table 5.4. Convolutional layers are parametrized by the input size, filter size, output (number of output feature map channels), padding, stride, and groups (number of filter groups). Padding refers to the number of pixels to implicitly add to each side of the input (zeros), while stride refers to the number of pixels the filter moves as it slides across the input. Filter groups refer to a Caffe function which can restrict the connections of each filter to a subset of the input, so the ith input group channels are connected only to the ith output group channels.

Table 5.4: Convolution results

AlexNet Convolutional Layers

Layer	Input	Filter	Output	Pad	Stride	Group	GPP (ms)	GPU (ms)	FPGA (ms)
Conv1	227*227	11*11	96	0	1	0	13.52 ± 0.1	2.89 ± 0.1	188.85 ± 0.6
Conv2	27*27	5*5	256	2	2	2	27.53 ± 0.2	3.80 ± 0.1	134.24 ± 0.4
Conv3	13*13	3*3	384	1	1	0	14.54 ± 0.1	2.41 ± 0.1	83.92 ± 0.3
Conv4	13*13	3*3	384	1	1	2	11.71 ± 0.1	2.10 ± 0.0	63.23 ± 0.3
Conv5	13*13	3*3	256	1	2	2	8.37 ± 0.0	1.47 ± 0.0	42.95 ± 0.2

FPGA resource utilization statistics for each convolution layer implementation are seen in Table 5.5.

Table 5.5: Convolutional Layer Resource Utilization

AlexNet Convolutional Layers

Layer	CU	FF	FF%	LUT	LUT%	DSP	DSP%	BRAM	BRAM%
Conv1	6	24258	4.4	21078	7.7	90	3.8	78	4.0
Conv2	6	19410	3.5	21474	7.8	102	4.3	48	2.5
Conv3	6	42108	7.6	35604	12.9	174	7.3	72	3.7
Conv4	6	37674	6.8	26640	9.7	132	5.5	48	2.5
Conv5	6	37698	6.8	26604	9.7	132	5.5	48	2.5

From the benchmark timings, we see that the GPU greatly outperformed the CPU and FPGA in terms of execution time, with the FPGA falling far behind the other two. This result is mainly due to the difficulty of created a highly optimized design for the FPGA given the SDAccel tool limitations with respect to a relatively complex algorithm such as convolution. A more optimized design using HLS and winograd convolution is seen in [12]. In terms of resource utilization, it is interesting to note that the heavier convolution design resulted in the ability to synthesize only a maximum of 6 compute units successfully. Though all of these convolution layers have different requirements, we notice that unsuccessful synthesis typically occurs during the routing and timing closure stages, meaning that the issue is likely related to the complexity of the logic rather than the amount of resources needed.

ReLU Activation

Benchmark timings for the ReLU activation layer implementations are provided in Table 5.6. ReLU layers are not typically considered an independent layer, but rather simply activations applied to every unit in the previous layer. However, given the Caffe implementation of ReLU, they are considered a layer in this context. Each ReLU layer is parametrized by the number units in the previous layer, which we denote as the input.

Table 5.6: ReLU results

AlexNet ReLU Layers

Layer	Input	GPP (ms)	GPU (ms)	FPGA (ms)
ReLU1	290400	0.24 ± 0.0	0.18 ± 0.0	1.78 ± 0.0
ReLU2	186624	0.15 ± 0.0	0.12 ± 0.0	0.97 ± 0.0
ReLU3	64896	0.05 ± 0.0	0.05 ± 0.0	0.41 ± 0.0
ReLU4	43264	0.04 ± 0.0	0.03 ± 0.0	0.31 ± 0.0
ReLU5	64896	0.05 ± 0.0	0.05 ± 0.0	0.41 ± 0.0
ReLU6	4096	0.01 ± 0.0	0.02 ± 0.0	0.11 ± 0.0
ReLU7	4096	0.01 ± 0.0	0.02 ± 0.0	0.11 ± 0.0

FPGA resource utilization statistics for each ReLU layer implementation are seen in Table 5.7.

Table 5.7: ReLU Layer Resource Utilization

AlexNet ReLU Layers

Layer	CU	FF	FF%	LUT	LUT%	DSP	DSP%	BRAM	BRAM%
ReLU1-7	8	33,768	6.1	28,520	10.4	0	0	256	13.2

From the benchmark timings, we see that the GPU outperforms both the GPP and FPGA in terms of execution time, with the GPP having a slight edge over the FPGA for all layer configurations considered. This result is interesting, as one would expect the FPGA to outperform the GPP for such a simple and inherently parallel algorithm. However, in practice we learned that for small computations, the overhead

of writing and reading from the FPGA is large enough to overcome the advantage of parallel computation. The benefit is only experienced for algorithms of substantial complexity. In terms of resource utilization, it should be noted that all layers used the same resources because they were all implemented using the same binary. For ReLU, we were able to simply compile one binary of maximal size (to accommodate the biggest layer ReLU1), and use partial computations to compute the other layers which divided easily into this size. This concept is one of fundamental importance to future work which may look to accommodate changes in hyper-parameters without recompiling an FPGA binary.

Pooling Layers

Benchmark timings for the max pooling layer implementations are provided in Table 5.8. Convolutional layers are parametrized by the input size, filter size, stride, and output (number of output feature map channels).

Table 5.8: Max pooling results

AlexNet Max Pooling Layers

Layer	Input	Filter	Stride	Output	GPP (ms)	GPU (ms)	FPGA (ms)
Pool1	55*55	3*3	2	96	1.08 ± 0.0	0.33 ± 0.0	1.98 ± 0.0
Pool2	27*27	3*3	2	256	0.71 ± 0.0	0.24 ± 0.0	1.14 ± 0.0
Pool3	13*13	3*3	2	256	0.18 ± 0.0	0.07 ± 0.0	0.46 ± 0.0

FPGA resource utilization statistics for each convolution layer implementation are provided in Table 5.9.

Table 5.9: Max Pooling Layer Resource Utilization

AlexNet Max Pooling Layers

Layer	CU	FF	FF%	LUT	LUT%	DSP	DSP%	BRAM	BRAM%
Pool1	6	11328	2.1	13620	4.9	54	2.3	336	17.3
Pool2	6	33840	6.1	60473	21.9	64	2.7	168	8.7
Pool3	6	57344	10.4	101284	36.7	36	1.5	84	4.3

From the benchmark timings, we see that the GPU outperforms both the GPP and FPGA in terms of execution time, with the GPP having a slight edge over the FPGA for all layer configurations considered. This result likely comes from a sub-optimal design of the FPGA device kernel, where it is likely that sequential operations are happening in place of parallel ones. In terms of resource utilization, the results are as expected for layers of increasing memory and logic requirements.

Local Response Normalization Layers

Benchmark timings for the local response normalization layer (LRN) implementations are provided in Table 5.10. LRN layers are parametrized by the input size, filter size, stride, and output (number of output feature map channels).

Table 5.10: Local response normalization layer results

AlexNet Local Response Normalization Layers

Layer	Input	Alpha	Beta	Local Size	Output	GPP (ms)	GPU (ms)	FPGA (ms)
Norm1	55*55	0.0001	0.75	5	96	10.80 ± 0.1	0.44 ± 0.0	6.78 ± 0.1
Norm2	27*27	0.0001	0.75	5	256	6.96 ± 0.1	0.36 ± 0.0	5.14 ± 0.1

FPGA resource utilization statistics for each LRN layer implementation are seen in Table 5.11.

Table 5.11: Local Response Normalization Layer Resource Utilization

AlexNet Local Response Normalization Layers

Layer	CU	FF	FF%	LUT	LUT%	DSP	DSP%	BRAM	BRAM%
Norm1	8	35680	6.5	42040	15.3	336	14.1	320	16.4
Norm2	8	35304	6.4	41608	15.1	336	14.1	80	4.1

From the benchmark timings, we see that the GPU outperforms both the GPP and FPGA in terms of execution time, with the FPGA outperforming the GPP for all layer configurations considered. This

result is in agreement with expectations, however since a few of the dependent mathematical operations needed to calculate LRN were performed sequentially on the FPGA, additional work would be required to improve this result. In terms of resource utilization, $Norm1$ required many more BRAM to accommodate the additional memory size resulting from handling larger input dimensions.

5.3 Model Benchmarking

The benchmarking results for the full AlexNet model are provided in Table 5.12. These benchmarks correspond to the original AlexNet deployment architecture, where only the dropout layers are ignored for this implementation. Dropout layers are especially difficult to implement on FPGAs given the heavy use of stochastic operations, and contemporary work tends to focus on the most common CNN layers such as convolution, pooling, normalization, and fully connected layers [50]. It should be noted that the program layers refer to the program layer extensions from FPGA Caffe introduced in this work, and denote the time needed to program the FPGA. Since Caffe does not include functionality to measure the time to program the GPU and GPP, the time was extracted by running empty kernels on these two architectures, and assuming this value for all layers.

Table 5.12: AlexNet Model Benchmarking

Layer Type	**GPP (ms)**	**GPU (ms)**	**FPGA (ms)**
Program1	0.00 ± 0.0	0.01 ± 0.0	142.56 ± 0.5
Conv1	13.52 ± 0.1	2.89 ± 0.1	188.85 ± 0.6
Program2	0.00 ± 0.0	0.01 ± 0.0	198.79 ± 0.5
Relu1	0.24 ± 0.0	0.18 ± 0.0	1.78 ± 0.0
Program3	0.00 ± 0.0	0.01 ± 0.0	142.12 ± 0.4
Norm1	10.80 ± 0.1	0.44 ± 0.0	6.78 ± 0.1
Program4	0.00 ± 0.0	0.01 ± 0.0	158.67 ± 0.5
Pool1	1.08 ± 0.0	0.33 ± 0.0	1.98 ± 0.0
Program5	0.00 ± 0.0	0.01 ± 0.0	141.39 ± 0.5

Continued on next page

Table 5.12 – *Continued from previous page*

Layer Type	**GPP (ms)**	**GPU (ms)**	**FPGA (ms)**
Conv2	27.53 ± 0.2	3.80 ± 0.1	134.24 ± 0.4
Program6	0.00 ± 0.0	0.01 ± 0.0	199.87 ± 0.6
Relu2	0.15 ± 0.0	0.12 ± 0.0	0.97 ± 0.0
Program7	0.00 ± 0.0	0.01 ± 0.0	136.74 ± 0.4
Norm2	6.96 ± 0.1	0.36 ± 0.0	5.14 ± 0.1
Program8	0.00 ± 0.0	0.01 ± 0.0	144.00 ± 0.4
Pool2	0.71 ± 0.0	0.24 ± 0.0	1.14 ± 0.0
Program9	0.00 ± 0.0	0.01 ± 0.0	182.93 ± 0.6
Conv3	14.54 ± 0.1	2.41 ± 0.1	83.92 ± 0.3
Program10	0.00 ± 0.0	0.01 ± 0.0	198.69 ± 0.6
Relu3	0.05 ± 0.0	0.05 ± 0.0	0.41 ± 0.0
Program11	0.00 ± 0.0	0.01 ± 0.0	171.07 ± 0.5
Conv4	11.71 ± 0.1	2.10 ± 0.0	63.23 ± 0.3
Program12	0.00 ± 0.0	0.01 ± 0.0	198.71 ± 0.6
Relu4	0.04 ± 0.0	0.03 ± 0.0	0.31 ± 0.0
Program13	0.00 ± 0.0	0.01 ± 0.0	175.08 ± 0.5
Conv5	8.37 ± 0.0	1.47 ± 0.0	42.95 ± 0.2
Program14	0.00 ± 0.0	0.01 ± 0.0	198.78 ± 0.6
Relu5	0.05 ± 0.0	0.05 ± 0.0	0.41 ± 0.0
Program15	0.00 ± 0.0	0.01 ± 0.0	161.57 ± 0.5
Pool3	0.18 ± 0.0	0.07 ± 0.0	0.46 ± 0.0
Program16	0.00 ± 0.0	0.01 ± 0.0	253.83 ± 0.9
Full6	47.39 ± 0.3	10.89 ± 0.3	23.43 ± 0.2
Program17	0.00 ± 0.0	0.01 ± 0.0	198.68 ± 0.6
Relu6	0.01 ± 0.0	0.02 ± 0.0	0.11 ± 0.0
Program18	0.00 ± 0.0	0.01 ± 0.0	224.42 ± 0.7
Full7	20.27 ± 0.2	4.72 ± 0.2	10.52 ± 0.1
Program19	0.00 ± 0.0	0.01 ± 0.0	198.58 ± 0.6

Continued on next page

Table 5.12 – *Continued from previous page*

Layer Type	**GPP (ms)**	**GPU (ms)**	**FPGA (ms)**
Relu7	0.01 ± 0.0	0.02 ± 0.0	0.11 ± 0.0
Program20	0.00 ± 0.0	0.01 ± 0.0	135.27 ± 0.4
Full8	4.92 ± 0.1	1.19 ± 0.1	2.49 ± 0.1
Total	**168.52 ± 1.3**	**33.38 ± 0.9**	**4130.98 ± 13.3**

From the benchmark timings, we see that the GPU is much faster than both the GPP and FPGA for deploying the full AlexNet model. It is important to note that the main reason why the FPGA is so slow in this context, is that the time to program the FPGA at each layer is very large compared to the GPU and GPP. Even assuming all operations on the FPGA were computed instantaneously, due to this issue the FPGA would still be much slower. To address this issue, we introduce pipeline layer benchmarks below.

5.4 Pipeline Layer Benchmarking

The motivation for pipeline layers was born out of the results seen in Table 5.12. When implementing FPGA support in Caffe, the logical starting point is to mimic the GPU approach, where before the execution of each layer the GPU is programmed with the appropriate device kernel and memory is synchronized with the host. However, the bottleneck in deploying full CNN models on FPGAs, using a GPU-based approach, is the time to program the FPGA between each layer. The logical improvement to this strategy is to combine all logic into a large single kernel inside of a binary container with sequential execution (non-pipeline approach), so that the FPGA only has to be programmed once between layers. However, this approach is non-ideal, as layers organized sequentially within a kernel cannot execute concurrently. The ideal solution to this problem is the pipeline approach, where multiple layers are combined into a single binary container, but exist as separate device kernels which can execute concurrently, and exchange data using local memory structures on the FPGA. To demonstrate this, we create an example of a small CNN with a convolution, ReLU, and max pooling layer. The benchmark results for this model using a GPU-based approach, a non-pipeline approach, and pipeline layer approach, are provided in Table 5.13.

Table 5.13: Experimental results for a small example CNN comparing the GPU-based, non-pipeline, and pipeline layer design approach. This model processes 5 feature maps for 1 input image of size 227×227 with 3 colour channels.

GPU-Based Approach	
Layer	Time (ms)
Program1	89.5
Conv	42.2
Program2	95.7
Relu	45.3
Program3	92.1
Pool	0.4
TOTAL	365.2

Non-Pipeline Approach	
Layer	Time (ms)
Program1	109.7
Conv+ReLU+Pool	72.8
TOTAL	182.5

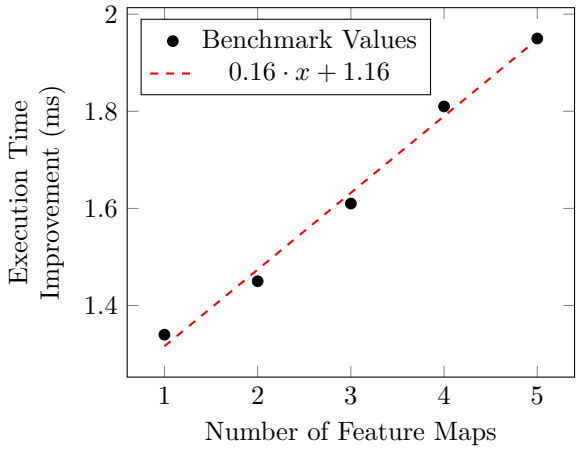
Pipeline Approach	
Layer	Time (ms)
Program1	109.7
Conv+ReLU+Pool	70.5
TOTAL	180.2

It is seen from the experiments that the pipeline approach completes the fastest, offering a substantial increase in performance compared with the GPU-based approach. The modest improvement over the non-pipeline approach is a result of scaling. Consistent with Figure 4.2, the effective increase in performance grows as more data is processed, and device kernels are able to overlap more. However, due to tight restrictions in local memory (FIFO) sizes and compute unit functionality in the current SDAccel version, we limited our investigation to an amount of data that could be processed using a single maximal size FIFO. The effect of scaling pipeline layers to include more feature maps is provided in Figure 5.1.

We see that as more feature maps are processed, the performance improvement of using pipeline layers increases linearly. Further testing of this result is restricted by the SDAccel tool, which limits our experiments to using only 5 feature maps (each work-group process a single feature map) with the pipeline layers. However, we can expect this trend to continue until the data size approaches the maximum allowed by SDAccel FIFOs (roughly equivalent to 10 feature maps of size 55×55). At this point the overhead of having multiple FIFOs may interfere with this linear trend, although testing could not be done at this time due to current limitations of FIFOs in SDAccel.

In justifying the practicality of pipelining layers, we argue that combinations of layer groups are very

Figure 5.1: Pipeline Layer Execution Time Improvement vs. Number of Input Images



predictable in practice. For example, In CNNs convolution layers are very commonly followed by pooling layers, with the most common variability coming from hyper-parameter selection. As such, it is practical to package common combinations of layer groups as a pipeline binary.

5.5 Power Consumption

Though FPGAs are typically considered low power devices compared to GPPs and GPUs, the FPGA Caffe framework is functionally heterogeneous, which complicates the way power consumption is measured and interpreted. The execution of host code, device code, memory synchronization, and device programming ensure that the framework is frequently changing between GPP and FPGA/GPU operation. To best capture the practical power consumption of this framework, and institute a fair comparison between the GPP, GPU, and FPGA, we measure power consumption of the host system through an electricity usage meter for deployment of the full AlexNet model. The configuration of the host system remains constant throughout these experiments, with only the accelerator configuration changing at runtime. We believe that the use of an electricity usage meter in these experiments gives a practical value which describes the systematic power consumption effect of using one accelerator in place of another. Due to this configuration, power consumption of the FPGA may seem high compared to work which uses similar FPGAs. However,

it is important to note that most related work uses FPGA cards powered externally (not through PCIe) which operate without the use of a host GPP [50]. These configurations are optimal for power consumption, but do not reflect real practical deep learning work flows which require frequent communication with a host GPP.

We measure power consumption for three configurations (GPP only, GPP + GPU, GPP + FPGA) as an average while deploying the entire AlexNet model for 100 iterations, and report values ± one standard deviation. The results for this are provided in Table 5.14. Theoretical averages are reported according to rough datasheet guidelines, and estimated for typical workloads [59, 60, 61].

Table 5.14: Power Consumption Measurements

	GPP (W)	**GPU (W)**	**FPGA (W)**
Theoretical Average	≈ 140.00	≈ 41.00	≈ 26.00
	GPP (W)	**GPP + GPU (W)**	**GPP + FPGA (W)**
AlexNet Average	154.20 ± 4.3	161.30 ± 4.4	157.10 ± 6.3

During experimentation it was noticed that power consumption for the FPGA spikes while the FPGA was being programmed, which led to an increase in average power consumption. Since non-pipeline layers were used for the full AlexNet measurements, pipeline layers would provide an improvement in power consumption. This effect is seen in Table 5.15, which uses the small CNN model from Table 5.13 which processes 5 feature maps using 1 input image of size 227 × 227 with 3 colour channels. It is interesting to note that during run-time these values varied less aggressively than the AlexNet measurements, due to the faster iteration speed relative to the measurement frequency of the electricity usage meter.

Table 5.15: Pipeline Layer Power Consumption

Model	**GPU-Based Approach GPP + FPGA (W)**	**Pipeline Approach GPP + FPGA (W)**
Conv+ReLU+Pool	157.05 ± 0.5	154.35 ± 0.5

It is useful to now re-evaluate our AlexNet model benchmarks to include a measure of power efficiency per layer. To do this, we calculate the MFLOPS/W (mega floating point operations per watt), a more meaningful value that encompasses a well rounded view of computational efficiency which includes both throughput and power consumption. The results are provided in Table 5.16.

Table 5.16: AlexNet Model Benchmarking

Layer	**GPP** (MFLOPS/W)	**GPP+GPU** (MFLOPS/W)	**GPP+FPGA** (MFLOPS/W)
Full	1.03 ± 0.0	4.32 ± 0.1	2.06 ± 0.0
Conv	11.60 ± 1.3	64.90 ± 12.3	1.91 ± 0.7
ReLU	1.31 ± 0.6	1.40 ± 0.9	0.16 ± 0.1
Pool	1.34 ± 0.1	3.76 ± 0.7	0.70 ± 0.2
Norm	0.08 ± 0.0	1.83 ± 0.3	0.13 ± 0.0

From the results, it is seen that the GPU performed best in terms of MFLOPS/W for all layers considered. The FPGA ranked second for the fully connected and LRN layers, while the GPP ranked second for convolution, ReLU, and pooling layers. It is interesting to note that for very simple layers, such as ReLU, the efficiency improvement of the GPU over the GPP was very small, indicating that for small simple layers the overhead of writing memory to the accelerator almost negates the pure computational speed improvement. While these results may seem underwhelming, they would be drastically improved by the ability to use pipeline layers for the full AlexNet model, which improves both execution time and power consumption. Future work for implementing full pipelined models is highly encouraged.

5.5.1 Economic Considerations

To interpret the power consumption results in a practical context, it is important to include a discussion of how economics affect these results. Table 5.17 shows the retail values (USD) of the GPP, GPU, and FPGA used for benchmarking.

Table 5.17: Benchmarking Hardware Costs (USD) [62, 63, 64]

GPP	GPU	FPGA
$ 996	$ 128	$ 3,200

Given the superior results of the GPU in terms of performance per Watt of power consumption and up front investment cost, these results would indicate that the GPU is currently a better practical choice than the FPGA in both the short and long term. However, these results are presented in a worst case scenario for FPGAs, as the small execution time of deployment (milliseconds) allows the minimal host GPP interaction to remain relatively significant, meaning the GPP power consumption significantly affects the overall average power consumption, as seen in Table 5.14. As a result, the average power consumption of the FPGA is only slightly better than the GPU for the AlexNet model. In training situations where the execution time is much longer (hours) the FPGA would run in isolation for much longer periods of time, greatly lowering the average power consumption, and likely providing a significant advantage over the GPU in terms of performance per Watt of power consumption. Coupled with future improvements in FPGA optimizations for deep learning, as well as reduced costs as FPGAs become more popular in consumer applications, the economic considerations may likely change in favour of FPGAs in the future, though the advantage will likely remain with GPUs for the present.

5.6 Summary

In this chapter, we discussed the results of an empirical benchmarking analysis of our work. We discussed the performance of the AlexNet layers, AlexNet model, and pipeline layers, in terms of execution time, resource usage, and power consumption. We observed that the GPU achieved the best performance in most performance metrics considered, which is expected given the maturity of the GPU-based optimizations used for comparison. However, we noticed that the FPGA outperformed the GPP for many criteria, which gives hope that further development of the FPGA-based optimizations presented in this thesis may lead to competitive performance with the GPU in the future.

Chapter 6

Conclusions

In this work, we introduce a practical design flow for deep learning on FPGAs, demonstrate this flow through the development of the FPGA Caffe framework, and support this framework with an empirical performance assessment. When introducing the design flow, we show how current technologies enable the support of FPGAs in common deep learning frameworks, and justify why this integration makes sense from the perspective of deep learning practitioners. To demonstrate this, we contribute to the development of the FPGA Caffe framework as a proof of concept, showing how this design flow can be used in practice. To address a few pitfalls of using FPGAs in this way, we introduce layer extensions to the FPGA Caffe framework which appeal to the needs of FPGAs, while not affecting the usability of the framework. Programming layers are used to ensure more control over the way the FPGA is programmed at runtime, and pipeline layers alleviate many of the problems associated with using FPGAs in a GPU-based framework. When evaluating performance, it is clear that the GPU is currently most efficient in terms of execution time and throughput, though the FPGA is best in terms of power consumption and eclipses GPP execution time performance in several CNN layers. We believe that, since this work is not as mature as equivalent GPU work, these results may change in the future as more effort is dedicated to improving performance of deep learning on FPGAs.

In closing, we believe that we have answered our objective question by showing that FPGAs can be used as a practical acceleration platform for deep learning. While, in its current state, this work does not convince deep learning practitioners to use FPGAs instead of GPUs, we believe that the directions which make this reality have been identified. We believe that this work is valuable in acting as a guide to future researchers who wish to continue efforts in this field, and encourage the future development of FPGA Caffe.

6.1 Future Work

The future of deep learning on FPGAs, and in general, is largely dependent on scalability. For these techniques to succeed on the problems of tomorrow, they must scale to accommodate data sizes and architectures that continue to grow at an incredible rate. Recent work has shown that increasing the depth of neural networks can result in increased performance, and neural networks with as many as 1000 layers have been trained successfully [9]. FPGA technology is adapting to support this trend, as the hardware is headed toward larger memory, smaller feature sizes, and interconnect improvements to accommodate multi-FPGA configurations. The Intel acquisition of Altera, along with the partnership of IBM and Xilinx, indicates a change in the FPGA landscape which may also see the integration of FPGAs in consumer and data center applications in the very near future. In addition, design tools will likely tend toward higher levels of abstraction and software-like experiences, in an effort to attract a wider technical range of users.

Increasing Degrees of Freedom for Training

While one may expect the process of training machine learning algorithms to be fully autonomous, in practice, there are tunable hyper-parameters that need to be adjusted. This is especially true for deep learning, where complexity of the model in terms of number of parameters is often accompanied by many possible combinations of hyper-parameters. The number of training iterations, the learning rate, mini-batch size, number of hidden units, and number of layers are all examples of hyper-parameters that

can be adjusted. The act of tuning these values is equivalent to selecting which model, among the set of all possible models, is best for a particular problem. Traditionally, hyper-parameters have been set by experience or systematically by grid search or more effectively, random search [65]. Very recently, researchers have turned to adaptive methods, which exploit the results of hyper-parameter attempts. Among these, Bayesian Optimization [66, 67] is the most popular.

Regardless of the method selected to tune hyper-parameters, current training procedures using fixed architectures are somewhat limited in their ability to grow these sets of possible models, meaning that we may be viewing the solution space through a very narrow lens. Fixed architectures make it much easier to explore hyper-parameter settings within models (e.g. number of hidden units, number of layers) but difficult to explore settings between models (i.e. different *types* of models) as the training of models which do not conveniently conform to a particular fixed architecture may be very slow. The flexible architecture of FPGAs, however, may be better suited for these types of optimizations, as a completely different hardware structure can be programmed and accelerated at runtime.

Low Power Compute Clusters

One of the most intriguing aspects of deep learning models is the ability to scale. Whether the purpose is to discover complex high level features in data, or to increase performance for data center applications, deep learning techniques are often scaled up across multi-node computing infrastructures. Current solutions to this problem involve using clusters of GPUs with Infiniband interconnects and MPI to allow high levels of parallel computing power and fast data transfer between nodes [1, 68]. However, as the workloads of these large scale applications become increasingly heterogeneous, the use of FPGAs may prove to be a superior alternative. The programmability of FPGAs would allow reconfiguration based on the application and workload, and FPGAs provide an attractive performance per watt that would lower costs for the next generation of data centers.

References

[1] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with COTS HPC systems," in *Proceedings of the 30th international conference on machine learning*, pp. 1337–1345, 2013.

[2] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[3] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[4] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A CPU and GPU math compiler in python," in *Proc. 9th Python in Science Conf*, pp. 1–7, 2010.

[5] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," tech. rep., IDIAP, 2002.

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

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[10] G. Lacey, G. W. Taylor, and S. Areibi, "Deep learning on FPGAs: Past, present, and future," *arXiv preprint arXiv:1602.04283*, 2016.

[11] G. Hinton, L. Deng, D. Yu, G. Dahl, A. rahman Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition," *Signal Processing Magazine*, 2012.

[12] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. W. Taylor, and S. Areibi, "Caffeinated FPGAs: FPGA framework for convolutional neural networks," 2016. In preparation.

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

[14] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing," in *Field-Programmable Technology (FPT), 2010 International Conference on*, pp. 94–101, IEEE, 2010.

[15] Y. Bengio and O. Delalleau, "On the expressive power of deep architectures," in *Algorithmic Learning Theory*, pp. 18–36, Springer, 2011.

[16] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015. Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved.

[17] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015. Published online 2014; based on TR arXiv:1404.7828 [cs.NE].

[18] Y. Bengio, "Learning deep architectures for AI," *Foundations and trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[19] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *CoRR*, vol. abs/1404.5997, 2014.

[20] Y. Bengio, I. J. Goodfellow, and A. Courville, "Deep learning." Book in preparation for MIT Press, 2015.

[21] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," *The Journal of physiology*, vol. 195, no. 1, pp. 215–243, 1968.

[22] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)* (G. J. Gordon and D. B. Dunson, eds.), vol. 15, pp. 315–323, Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011.

[23] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[24] D. F. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013.

[25] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," in *Parallel Processing (ICPP), 2011 International Conference on*, pp. 216–225, IEEE, 2011.

[26] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From opencl to high-performance hardware on FPGAs," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 531–534, IEEE, 2012.

[27] "Xilinx SDAccel." `http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`, 2015.

[28] "Khronos OpenCL." `https://www.khronos.org/opencl/`, 2016.

[29] I. G. Y. Bengio and A. Courville, "Deep learning." Book in preparation for MIT Press, 2016.

[30] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, and Others, "TensorFlow: Large-scale machine learning on heterogeneous systems, 2015," *Software available from tensorflow. org*, vol. 1, 2015.

[31] C. E. Cox and W. E. Blanz, "Ganglion-a fast field-programmable gate array implementation of a connectionist classifier," *Solid-State Circuits, IEEE Journal of*, vol. 27, no. 3, pp. 288–299, 1992.

[32] J. G. Eldredge and B. L. Hutchings, "Density enhancement of a neural network using FPGAs and run-time reconfiguration," in *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*, pp. 180–188, IEEE, 1994.

[33] P. Lysaght, J. Stockwood, J. Law, and D. Girma, "Artificial neural network implementation on a fine-grained FPGA," in *Field-Programmable Logic Architectures, Synthesis and Applications*, pp. 421–431, Springer, 1994.

[34] A. Pérez-Uribe and E. Sanchez, "FPGA implementation of an adaptable-size neural network," in *Artificial Neural NetworksICANN 96*, pp. 383–388, Springer, 1996.

[35] Y. Taright and M. Hubin, "FPGA implementation of a multilayer perceptron neural network using VHDL," in *Signal Processing Proceedings, 1998. ICSP'98. 1998 Fourth International Conference on*, vol. 2, pp. 1311–1314, IEEE, 1998.

[36] J. Zhu and P. Sutton, "FPGA implementations of neural networks–a survey of a decade of progress," in *Field Programmable Logic and Application*, pp. 1062–1066, Springer, 2003.

[37] A. R. Omondi and J. C. Rajapakse, *FPGA implementations of neural networks*, vol. 365. Springer, 2006.

[38] A. Canas, E. M. Ortigosa, E. Ros, and P. M. Ortigosa, "FPGA implementation of a fully and partially connected MLP," in *FPGA Implementations of Neural Networks*, pp. 271–296, Springer, 2006.

[39] K. Paul and S. Rajopadhye, "Back-propagation algorithm achieving 5 gops on the virtex-e," in *FPGA Implementations of Neural Networks*, pp. 137–165, Springer, 2006.

[40] R. Gadea, J. Cerdá, F. Ballester, and A. Mocholí, "Artificial neural network implementation on a single FPGA of a pipelined on-line backpropagation," in *Proceedings of the 13th international symposium on System synthesis*, pp. 225–230, IEEE Computer Society, 2000.

[41] A. W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study," *Neural Networks, IEEE Transactions on*, vol. 18, no. 1, pp. 240–252, 2007.

[42] S. Vitabile, V. Conti, F. Gennaro, and F. Sorbello, "Efficient MLP digital implementation on FPGA," in *Digital System Design, 2005. Proceedings. 8th Euromicro Conference on*, pp. 218–222, IEEE, 2005.

[43] E. Ordoñez-Cardenas and R. d. J. Romero-Troncoso, "MLP neural network and on-line backpropagation learning implementation in a low-cost FPGA," in *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pp. 333–338, ACM, 2008.

[44] J. Misra and I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress," *Neurocomputing*, vol. 74, no. 1, pp. 239–255, 2010.

[45] K. P. Lakshmi and M. Subadra, "A survey on FPGA based MLP realization for on-chip learning," *Int. J. Sci. & Eng. Res*, vol. 4, no. 1, pp. 1–9, 2013.

[46] A. Gomperts, A. Ukil, and F. Zurfluh, "Development and implementation of parameterized FPGA-based general purpose neural networks for online applications," *Industrial Informatics, IEEE Transactions on*, vol. 7, no. 1, pp. 78–89, 2011.

[47] J. Cloutier, S. Pigeon, F. R. Boyer, E. Cosatto, and P. Y. Simard, "VIP: An FPGA-based processor for image processing and neural networks," in *microneuro*, p. 330, IEEE, 1996.

[48] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, 2015.

[49] "Microsoft Research: catapult," 2015.

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

[51] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 273–284, ACM, 2010.

[52] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 247–257, ACM, 2010.

[53] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An FPGA-based processor for convolutional networks," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pp. 32–37, IEEE, 2009.

[54] M. Peemen, A. Setio, B. Mesman, H. Corporaal, *et al.*, "Memory-centric accelerator design for convolutional neural networks," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pp. 13–19, IEEE, 2013.

[55] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pp. 53–60, IEEE, 2009.

[56] A. Savich, M. Moussa, and S. Areibi, "A scalable pipelined architecture for real-time computation of MLP-BP neural networks," *Microprocessors and Microsystems*, vol. 36, no. 2, pp. 138–150, 2012.

[57] "Opencl on FPGAs for GPU programmers." `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-201406-acceleware-opencl-on-fpgas-for-gpu-programmers.pdf`, 2015.

[58] "GPU vs FPGA performance comparison," tech. rep., Berten Digital Signal Processing, 2016.

[59] Intel, *Intel Xeon Processor E5 v2 Family: Datasheet*, 3 2014. Vol. 2.

[60] NVIDIA, *Quadro K600 Datsheet*, 10 2013. Rev. 1.0.

[61] Alpha Data, *Alpha Data ADM-PCIE-7V3 User Guide*, 6 2016. Rev. 1.4.

[62] "Intel ARK." `http://ark.intel.com/products/75792/Intel-Xeon-Processor-E5-2637-v2-15M-Cac50-GHz`, 2016.

[63] "Futuremark Graphics Cards." `http://www.futuremark.com/hardware/gpu/NVIDIA+Quadro+K600/review`, 2016.

[64] "Alpha Data US Store." `https://usastore.alpha-data.com/ADM-PCIE-7V3?search=7v3`, 2016.

[65] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 281–305, 2012.

[66] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of Bayesian optimization," *Proc. IEEE*, vol. 104, pp. 148–175, Jan. 2016.

[67] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, pp. 2951–2959, 2012.

[68] H. Ma, F. Mao, and G. W. Taylor, "Theano-MPI: a Theano-based distributed training framework," in *Euro-Par Workshop on Unconventional High Performance Computing*, 2016.

Appendix A

FLOP Calculations

To calculate the number of floating point operations (FLOP) in each AlexNet layer, let us first define shorthand for describing the sizes of the input, output, and filters:

Input: IN_{size} = number of input units (N_i) × channels (N_c) × input width (W_i) × input height (H_i)
Filter: $FILT_{size}$ = number of filter units (N_f) × filter width (W_f) × filter height (H_f)
Output: OUT_{size} = number of output units (N_o) × output width (W_o) × output height (H_o)

For these calculations, filters are used to describe both convolution and pooling filters. For these layers, the output size can be calculated using layer-specific choices of stride, padding, filter width, and filter height.

Fully Connected Layers

Since fully connected layers take as input vectors as opposed to images, they are parametrized by the number of input and output units. In Caffe, input and output blobs would have the width and height set to 1. For each iteration in the fully connected layer algorithm both a multiply and sum operation are used (2 FLOP per innermost loop iteration):

$$\text{FLOP}_{\text{full}} = 2 \times N_i \times N_o \tag{A.1}$$

Convolutional Layers

Similar to fully connected layers, for each iteration in the convolutional layer algorithm, both a multiply and sum operation are used (2 FLOP per inner most loop iteration):

$$\text{FLOP}_{\text{conv}} = 2 \times N_i \times OUT_{size}(N_c \times W_f \times H_f + 1) \tag{A.2}$$

ReLU Activation

Each activation in ReLU layers is performed on the input, using 2 FLOP for comparing and setting the output value:

$$\text{FLOP}_{\text{relu}} = 2 \times N_i \tag{A.3}$$

Pooling Layers

Since each max operation involves $n-1$ comparisons, we can assume that $\max\{x_1, x_2, ..., x_n\}$ accounts for $n-1$ FLOPs:

$$\text{FLOP}_{\text{pool}} = (W_f \times H_f - 1) \times IN_{size} \tag{A.4}$$

Local Response Normalization Layers

Finally, in LRN layers we perform 2 FLOP (squaring and accumulating the activation) for each pixel on each image within a given local size N_l, and perform 4 FLOP (to handle α, β, k, and division) for each pixel in the input image:

$$\text{FLOP}_{\text{lrn}} = (W_i \times H_i \times N_l \times 2) + (W_i \times H_i \times 4) \tag{A.5}$$