ETH ZÜRICH

MASTER'S THESIS

A Deep Learning Library for FPGAs using OpenCL

Author:
Houssam NAOUS

Supervisor:
Dr. Torsten HOEFLER

A thesis submitted in fulfillment of the requirements for the degree of Masters of Science in Computer Science

in the

Scalable and Parallel Computing Lab
Computer Science

August 25, 2018

Declaration of Authorship

I, Houssam NAOUS, declare that this thesis titled, "A Deep Learning Library for FPGAs using OpenCL" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

ETH ZÜRICH

Abstract

Computer Science
Computer Science

Masters of Science in Computer Science

A Deep Learning Library for FPGAs using OpenCL

by Houssam NAOUS

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

Chapter 1

Introduction

1.1 Deep Learning and Applications

1.1.1 History

The first artificial neural network traces back to 1958 where it was first conceived by psychologist Frank Rosenblatt [35]. It was called the perceptron and it was meant to model the way a human brain adapts to inputs from the external world to learn binary classification tasks. At some point someone realized that this model could be useful in pattern matching tasks. The artificial neural network is organized into layers of a single threshold logic unit that models a single neuron in the human brain. In the early days, these models were constructed physically and later on were simulated on a single computer [30]. Nowadays, learning tasks are distributed and coordinated on multiple machines to achieve a single learning task [14]. The primitive perceptron developed into a structure of layers organized and separated by non-linearities. In theory, the "Universal Approximation Theorem" states that a multi-layer perceptron with one hidden layer containing a finite number of neurons can approximate any continuous function under some assumptions on the activation function [12].

1.1.2 Learning in Artificial Neural Networks

Towards the end of 1986, Hinton's paper titled "*Learning representations by back-propagating errors*"[37] was published and it introduced the usefulness of an algorithm called back-propagation that can train an artificial neural network that is organized into layers. It proved more useful than the previously know perceptron-convergence algorithm [46] and by the end of the 1980s many scientific institutes adopted the use of neural networks and utilized them to solve many tasks [33]. Unlike standard algorithms that rely on conditional procedures and hand-crafted logic, the artificial neural network if designed properly is robust to noise and can adapt to those pattern matching tasks [45]. Artificial neural networks are exposed to thousands or millions of that are forward propagated through the weights in each of the layers. In-between each layer non-linearities are introduced and the output is compared to a specific target encoding of the output labels. From that a loss can be calculated and using a process called backpropagation [37], starting with the output layers, the network readjusts its weights to better match the target output, specifically reinforcing the connections that contribute to a correct output label.

1.1.3 Deep Learning

As computing resources were cheaper and more available and after the numerous improvements in computer hardware and architecture, scientists were able to simulate more complex networks with more neurons and deeper layers [17]. In fact, it was even proved that deeper networks with less neurons per layer proved more useful than the shallow networks [42], thus the concept of deep learning was popularized. Deep learning is only a subset of the broader concept of machine learning which consists of supervised, semi-supervised, and unsupervised learning tasks. It has proven itself useful in applications related to computer vision, speech, recognition, finance, and many others. The hype over deep learning increased even more when these networks were trainable on Graphical Processing Units (GPUs) which are capable of performing floating point operations on hundreds and thousands of cores in parallel [34]. This kind of parallelization decreased training time for these networks drastically and soon enough the suitable frameworks were developed and popularized [1, 4]. Researchers were then able to utilize those hardware for training neural networks with more data and experiment with more sophisticated network models and architectures [21, 34, 17].

1.2 Modern FPGAs

1.2.1 The Case for FPGAs

The market for Field-Programmable Gate Arrays (or FPGAs) has been increasingly growing and is expected to reach $12.98 billion by 2023 with a compound annual growth rate of 9.0% [1]. The demand for FPGAs was sparked by the need for high-throughput and low latency applications in industries such as aerospace, finance, and security. FPGAs are integrated circuits that are manufactured in a way such that they can be configured after production [10]. Using hardware descriptive languages (HDL) a hardware engineer can build a specific circuit and transfer it to the FPGA where it can reconfigure itself and rewire to implement a given circuit design. They offer a cheaper alternative to high-performing and specialized ASICs as they require less recurring engineering/manufacturing costs and less time to market which is necessary to thrive in this fast-paced economy. They offer a whole new dimension of customization in which complex instruction pipelines can be designed and implemented as opposed to the fixed instruction set architecture of a microcontroller or a generic CPU [25].

1.2.2 FPGAs vs GPUs

Application scientists have favored in the last couple of years the use of GPUs to accelerate deep learning tasks [34]. The GPUs architecture lends itself perfectly to perform parallel floating point computations needed to compute and train the networks. Moreover, what has lead to the GPUs more widespread use is a well defined programming model and tool-sets that are easily adopted by software programmers [13]. Dealing with those needs minimal experience in hardware architecture and applications have been parallelized and scaled massively. FPGAs on the other hand can offer higher power efficiency for the same computational workload as that of a GPU

[1] https://globenewswire.com/news-release/2017/11/29/1210234/0/en/Global-Field-Programmable-Gate-Array-FPGA-Market-to-Reach-USD-12-989-1-million-by-2023.html Last Accessed : 22/08/2018

and are intrinsically parallel devices [32]. However it's speed has not yet caught up with it's accelerator counterpart. Power efficiency is a major concern for large-scale applications operating in data centers. For that we have seen most recently both Microsoft [2] and Amazon [3] have incorporated FPGAs into their existing cloud computing infrastructure both for internal use against their data and as a service offered to clients wishing to utilize the power of reconfigurable architectures. Even though FPGAs up to date have only proven to be more power efficient than GPUs [25, 32], simulations and projections done at Intel Corporation predict that the upcoming generation of FPGAs will also compete with GPUs in terms of performance [32]. The projections show that the new Intel Stratix 10 is estimated to achieve 60% higher performance and 2.3 times less power consumption than the Titan X GPU. It is also important to note that the future of deep neural networks (DNNs) have resorted to fixed point computations and lowered precision up to the point of binary values as in Binarized Neural Networks [19]. All of these innovations have lead to irregular types of parallelism in which FPGAs excel at compared to GPUs. GPUs can only operate on a fixed set of data types and thus the trend towards lowering precision tips the scale of performance towards FPGAs [19, 38, 32]. The gap in performance between FPGAs and GPUs is getting smaller and thus it is necessary to update the toolset and design workflows in designing FPGA applications to keep up and make them more accessible for developers to be able to experiment and test.

1.3 High-Level Synthesis and OpenCL

1.3.1 FPGA Workflow

One of the main reasons that has lead to the slow adoption of FPGAs into commercial applications is the steep learning curve and technical background in hardware design required to be able to design and deploy applications. The usual workflow differs from that of a typical CPU application, however analogies can be drawn to bridge the gap between the two classes of applications. While designing a CPU application starts with a high-level programming language like C++ or Python, designing an FPGA circuit requires the use of hardware descriptive languages. The two most popular hardware descriptive languages are Verilog and VHDL [47]. Some FPGA design tools also offer the designer graphical user interfaces to draw schemas by dragging and dropping boxes. The latter, however, doesn't scale for larger projects and makes it harder to collaborate within a team. HDLs are dataflow programming languages that allow for broader descriptions of how digital circuits can communicate and execute logic in ways where other procedural languages like C fall short. An FPGA application designer also has additional stages in the design cycle where behavioural simulation as well as timing simulations and functional simulations of the design. After that there is a synthesis, placement, and fitting steps in which the designer hands his design to a piece of software that performs optimizations and tries to materialize the design in terms of a binary file which can be loaded onto the FPGA. Following that the developer runs also more tests on a development board and makes sure to fix any remaining bugs or errors. Another difficulty is that FPGA designs are not portable and thus certain parameters always need to be tuned to adapt to different boards with varying configurations. Each FPGA comes with a

[2] https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/ Last Accessed: 22/08/2018

[3] https://aws.amazon.com/ec2/instance-types/f1/ Last Accessed: 22/08/2018

different operating specification and different resource blocks [4], so in order to maximize utilization, the developer is expected to customize their design for each and every board.

1.3.2 High-Level Synthesis

High-Level Synthesis (HLS) is not a new invention at all. It has always co-existed with FPGAs. From initial research into HDLs, HLS tools have advanced into C-based dataflow programming paradigms nowadays. We see the level of abstraction rising from gate level, to register-transfer level, into algorithmic level synthesis [16, 29]. HLS is mainly motivated by the need to abstract hardware design to application programmers who should focus on designing and optimizing algorithms regardless of the underlying hardware. This isolation leaves hardware designers with the responsibility of optimizing intermediate representations of algorithms into synchronized logic blocks. Modern HLS tools begin by compiling the input specifications. Many code optimizations like code folding and dead-code elimination are carried-out to get a near-optimal input specification [16]. A control dataflow graph (CDFG) is created that parses the specification into a graph formed of nodes which are the basic blocks and connections that represent control dependencies between those blocks. The results is a register-transfer level (RTL) representation. It consists of a datapath (memory elements, interconnects, and functional units) and a control path. The controller is a finite-state machine that coordinates the flow of elements and operations on the datapath. The RTL representation is then verified to make sure it meets timing constraints and transformed into a gate-level representation that is then synthesized onto an FPGA according to a board specification file.

1.3.3 Xilinx and Intel

We have briefly motivated the main goal of HLS in abstracting the process of hardware design and mapping onto FPGAs. The two main manufacturers and technology leaders in the FPGA market are Xilinx and Intel (after the acquisition of Altera in 2015). Both companies have shifted their tool-sets to favor higher-level abstractions for synthesis but they have taken slightly different approaches. We note that Xilinx's development workflow favors more the hardware engineer by giving more control for them to view/modify designs and control most of the nuts and bolts that transform their applications into hardware [48]. Altera's tools however favor the software developer wishing to leverage hardware accelerators to achieve higher throughput for their application. The Intel FPGA developer is provided with a programming manual that suggests code improvement so that a better hardware design is generated. Both companies have adopted the use of HLS tools that can transform code in behavioural C/C++ and OpenCL descriptions into bitstreams [48, 22].

1.3.4 OpenCL for FPGAs

OpenCL™(Open Computing Language) is the open standard for cross-platform parallel programming that is a subset of the C standard [40]. OpenCL provides a programming model that fits the GPU architecture perfectly and is able to exploit parallelism through vectorized operations. GPUs are able to perform vectorized operations and have higher memory bandwidth than CPUs, enabling them to achieve

[4] https://www.intel.com/content/www/us/en/fpga/devices.html Last Accessed 23/08/2018

high throughput by doing parallel floating point operations. The challenge in adopting OpenCL for FPGAs is being able to not only vectorize operations but to also create efficient pipelines that fully utilize the resources available on the board. For that the Intel OpenCL SDK [22, 25] performs those optimizations and allows the user to use pre-defined pragmas in order to adapt OpenCL for FPGAs. This again beats the goal for portability across platforms when different customizations have to be introduced for FPGAs, however pipeline parallelism and complex data-flow instructions prove to be an advantage and a necessary feature that should be exploited on FPGAs [3, 25]. It is also worth noting that effort for optimizing the same OpenCL kernel differs between FPGAs and GPUs differs a lot putting FPGAs at a slight disadvantage. The reason is that with GPUs, the developer looks for the best mapping into the fixed architecture, while for FPGAs the developer guides the compiler into finding the best control and memory architecture for the given task. Moreover, compiling OpenCL kernels for FPGAs takes much longer than on GPUs as more board-specific optimizations can be done and because FPGAs allow for a broad design-exploration space.

1.4 Related Work

An implementation called "PipeCNN" [44] using OpenCL has explored the power of pipelining neural network layers to lower the overall memory bandwidth requirement in between network layers. The implementation was able to achieve a maximum throughput of 12.8GB/s on the Altera Stratix-5 board. The implementation however only covers convolution layers and fully connected layers. A single matrix-multiplication based kernels implements both of the convolution and fully connected layers and pipelines them with a pooling operation. In this implementation, full inter-layer communication is done by communicating through global memory. Local response normalization (LRN) which is done after the pooling layer is not pipelined directly and also communicated through global memory. The implementation, even though it utilizes the full memory bandwidth of the board, requires a lot of overhead for inter-layer communication and can be further reduced by pipelining more layers together.

The work of DiCecco et. al [15] utilizes the Xilinx SDAccel [5] toolset for optimizing neural network designs. They implement an FPGA backend for the popular neural network framework Caffe [23].This methodology makes use the already existing testbenches in Caffe for verifying correctness of the FPGA implementations. The authors run experiments using modern deep neural nets such as Alexnet, GoogleNet, and VGG-16 and achieve a maximum throughput of 50 GFLOPS across the 3x3 convolutions on the Xilinx Virtex 7. The authors also implement convolution using the Winograd [26] minimum filtering algorithm. This filtering scheme minimizes the number of multiplications required due to overlapping intermediate results in overlapping filter computations. The work lays the stepping stones and mentions the challenges in integrating FPGAs as accelerators for popular DNN frameworks like Caffe such as long reprogrammability times (100-400ms for FPGA vs 0,001ms-0.005ms for a GPU) and thus different types of parallelism should be exploited to fill in the gaps. FPGA implementations also require offline compilation and several

[5] /urlhttps://www.xilinx.com/products/design-tools/software-zone/sdaccel.html Last Accessed: 23/08/2018

vendor specific attributes to achieve optimal performance. The results are not impressive showing that GPU performance is still higher and the framework is still far from integrating the different layers of a DNN other than convolutions.

Zhang et. al.[**Zhang:2015**] were able to achieve 61.2 GFLOPS under 100 MHz on a VC707 FPGA. The main contribution of this work is an analysis framework which takes into consideration both the computing resources and the memory bandwidth provided by the board to guide the design space exploration phase. They balance out loop unrolling factors and loop tiling methods to balance the tradeoffs of using compute resources and memory bandwidth. The work however only focuses on the inference phase and lacks the analysis of the training phase of a CNN which could take days or weeks for a single learning task. They also focus on single layer optimizations for and not on multiple layer optimizations. This is important as compute intensive layers like convolutions can be balanced out with bandwidth hungry layers like the fully connected layers to achieve better performance.

1.5 Motivation

This work aims to leverage the benefits of OpenCL for programming FPGAs and target implementations of modern deep neural networks. The field has gained a lot of traction and so far the support for FPGA backends for accelerating neural network computations are still research based and experimental. It is also becoming increasingly important to utilize FPGA's configurable circuits for latency-sensitive and real-time DNN applications such as autonomous driving. By using FPGAs, neural networks can be accelerated and energy efficient by utilizing pipeline parallelism. Asides from that, an additional goal is not only to create networks for inference but to also accelerate training of deep neural networks on FPGAs. For that we use the Lenet network as a case study and proof of concept and implement minibatch gradient descent for training this network. We also aim to bridge the gap between research and application, so we have created a framework in Python that is able to go from open source model definitions like ONNX into material FPGA implementations. The development framework is easily extensible with modular components that also allow for individual customization and research into novel ways of accelerating the layer computations, backpropagation, and full network pipelining as a whole.

1.6 Outline of Next Sections

- Chapter 2 explains the algorithms and terminology in deep learning.
- Chapter 3 explains the toolset and hardware implementation of deep neural networks on FPGAs.
- Chapter 4 analyzes and discusses the results of the experiments of running the implementation on the FPGA.
- Chapter 5 servers as a primer and best practices in using OpenCL for FPGAs.
- Chapter 6 contains the concluding remarks and future direction of work.

Chapter 2

Deep Neural Networks and Parallelism

2.1 Deep Learning Applications

Deep learning and the progress made in the field have all fueled its integration into many applications of daily life [3]. In the early days, the results of machine learning were dependent on the quality and informativeness of the features fed into the learning algorithm [12]. With the development of deep neural nets, the machine itself can abstract raw data and learn complex features that become much more informative than the hand-engineered ones. Modern architectures are able to process and incoming stream of images, video, or speech and learn to classify those raw data and solve complex problems in image segmentation and speech recognition [27, 17, 21]. What is also important is that the value gained by those architectures increases as more data and more computational resources are presented to the network for training. Learning can be supervised, semi-supervised, and unsupervised [27]. Supervised learning is a problem of finding the best mapping between a set of input data X and a set of output labels or values Y. The algorithm proceeds in approximating the mapping function $f : X \rightarrow Y$ by observing both the inputs and the labels and learning stops when an acceptable approximation of f is found. Supervised learning can be split into two categories; classification when the output variable is a given label as in objects : *"table"* , *"chair"* , or *"motorcycle"*, and regression when the output variable is a real-value such as *"temperature"* or *"dollar value"*. The goal of unsupervised learning on the other hand is to discover a data model or structure of the given data. There are no given labels or output variables and the goals is to learn something useful about the data. Unsupervised learning is split also into two categories; clustering by splitting the data into groups having similar attributes, and association where we want to discover rules that describe a large subset of the observed data (ex. People who work at *"X"* tend to buy product *"Y"*). Semi-supervised learning is situated somewhere in between supervised and unsupervised learning where only a subset of the data is labeled. Solving these problems requires both supervised techniques to label the unlabelled data and feed back in the labels for training, or the data can be used to uncover structure. Our work focuses more on supervised learning as the majority of deployed applications use supervised learning. We attempt to accelerate and develop an FPGA accelerator for training deep neural networks.

2.2 Convolutional Neural Networks

Convolutional Neural Networks have achieved many successes in multidimensional data than can be represented as arrays. A typical image can be represented as a

three-channeled (Red, Green, and Blue) two-dimensional array of varying pixel intensities. A convolutional neural net is mainly composed of two stages that make sense of structured grid-like data to accomplish a learning task such as classification, regression, or even dimensionality reduction. An initial stage of convolution operators that pass a window filter over the image to extract an output feature map as in a discrete convolution operator. The convolution operation was inspired by a cat's visual cortex and how neurons in the brain are triggered by certain shapes like lines in the seen image [20]. The intuition behind the convolution operation as a feature extraction method is that nearby pixels exhibit similarities and show high correlations. Also the image patches extracted in the convolution windows which represent concepts can appear anywhere elsewhere in the image. The convolution layers are usually followed by non-linear activations and then pooling operators. The role of pooling layers is to merge the similar nearby features into one to decrease the sensitivity of the network to the position of the extracted feature [42]. The second stage of a CNN is a series of fully connected layers ultimately terminating at an output layer.

2.2.1 Case Study : LeNet-5

The Lenet [28] is considered to be the first demonstration of a convolutional neural network and was proposed by LeCun et. al [28] in 1998. Its architecture has shown record accuracy in classifying digits and was deployed commercially for identifying characters on personal and business checks. Le Cunn's work was the first to demonstrate the benefit of brute-force numerical tricks in convolutions and that these tricks can be more useful than hand-engineering traditional feature engineering techniques. Also compared to traditional neural networks, the convolution layers are more robust as the extracted features are shift and translation invariant. This proved to be efficient since handwritten characters differ in slant and scale from one person's handwriting to another's and it was hard back then to perform feature preprocessing to documents such as normalization and centering handwritten characters.

Architecture

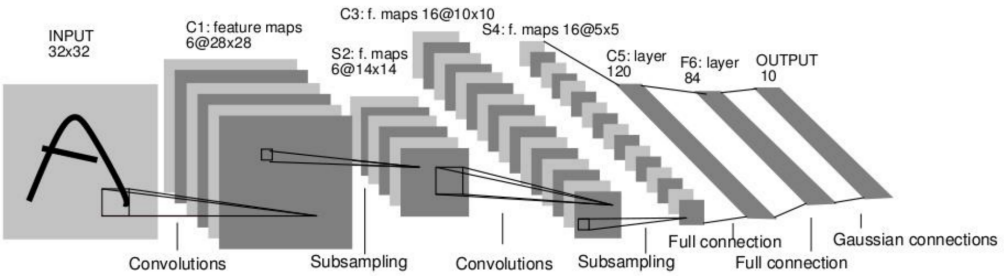

FIGURE 2.1: LeNet-5 Architecture, a convolutional neural network used for character recognition. [28]

The Lenet-5 is comprised of 7 layers (excluding the input layer), all of which contain trainable parameters. The input is a centered 32x32 image which represents the raw data that is fed into the network. It is followed by a convolutional layer and a subsampling layer. The first convolutional layer C1 produces an output of 6 distinct

feature maps. Each feature map is obtained by sliding a 5x5 filter along the input image and computing a weighted sum of the pixels inside the window. This leads to shrinking the image by 4 pixels along both dimensions, and 6 features are obtained by passing 6 different weights for the filters. A bias is added to the weighted sum of pixels and it is squashed by the sigmoid function before feeding into the output feature map.

The convolutional layer is then followed by a subsampling layer with a window size of 2x2. The four feature maps elements in C1 are added, multiplied by a trainable coefficient and then a trainable bias is added. The six feature maps of C1 are halved in size along both dimensions and the resulting feature maps at S2 are six 14x14 arrays.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

FIGURE 2.2: Each column indicates which feature maps in S2 are combined by the units in a particular feature map of C3. [28]

Similarly the feature maps at S2 are followed by another convolution and another subsampling layer C3 and S4. The outputs of S2 however are not connected all-to-all to the third convolution instead only the marked slots in the table 1 below shows that only some combinations of the feature maps in S2 are connected to C3. The purpose is to avoid overfitting to the full S2 features by breaking the symmetry and hopefully discovering different feature abstractions from the different set of inputs [28]. It also serves to decrease the amount of trainable parameters in the model.

The convolution stage (C1 $\rightarrow$ S4) is followed by two fully connected layers C5 and F6 of sizes 120 and 84 neurons respectively. Note that the reason C5 is labeled as a convolution layer (even though it performs the role of a fully connected layer) is that if the network were scaled for bigger inputs, the output feature map for C5 would be larger than 1x1.

The last fully connected layer is fed into the output which is composed of Euclidean Radial Basis (RBF) [5]. It computes the euclidean distance between the input vector and a trainable parameter vector (eq.1). The model was trained on a set of 60,000 images and achieved a minimum test-error of 0.7% competing with all of the other neural network models at the time. Some of the input training images were translated slightly, and rotated by up to $\pm 30^{\circ}$ to increase the network's robustness by making it slightly more translation and rotation invariant.

Even though the Lenet is outdated and has been replaced by many other neural networks, we choose to this network as a prototype to build our DNN framework due to the simplicity of the implementation. We implement it not for its usefulness but rather more to guide our study into optimizing layer implementations and thinking about methods to improve throughput and achieve pipelining between the layers.

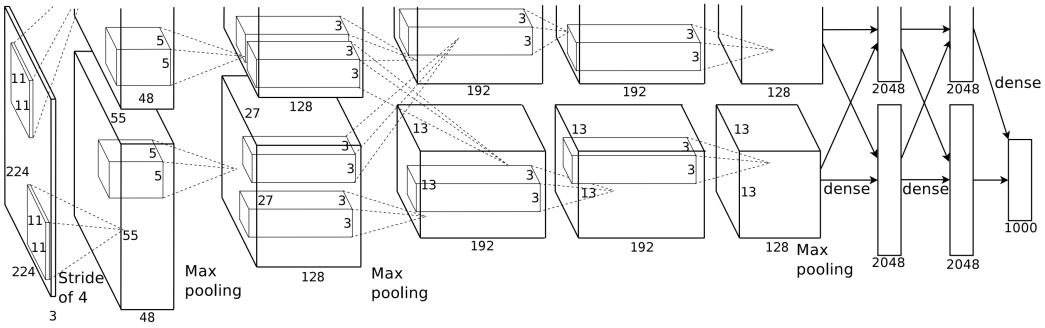

FIGURE 2.3: Alexnet Architecture [24]

2.2.2 Modern Architectures

AlexNet

AlexNet [24] won the ImageNet Large-scale Image recognition competition in 2012. It follows the Lenet's approach of staging some convolutional layers back-to-back and then eventually terminating with fully connected layers to perform the classification. It was able to achieve 26.25 top-5 error in the task of classification of three-channeled 224x224 images into 1000 classes. The network was trained on a GPU [24] and used local response normalization layers. In addition to that the authors used dropout which is a technique against overfitting in a network [39]. It involves randomly picking out neuron connections and setting them to zero thus decreasing the number of trainable parameters in the model. What also differs from LeNet is the use of Rectified Linear Units (ReLU) activations and maxpooling [24] for the subsampling layers.

ResNet

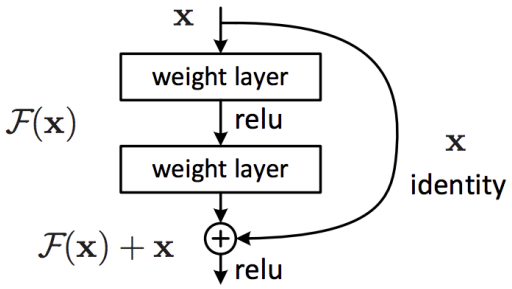

FIGURE 2.4: A basic building block in residual networks showing the identity shortcut [17]

ResNet [17] is one of the modern networks that sacrifices breadth for depth. Training deeper network is harder because of a problem discovered by Hochreiter et al. [18] of vanishing gradients. The authors of ResNet augment the network with shortcut connections that act as identity layers and enable training with respect to residuals instead of the original input values. Using this trick it was possible to train deeper networks reaching up to more than 150 layers.

2.3 Training and Backpropagation

The above networks all try to achieve the goal of properly approximating a mapping function between the dataset inputs and the given labels. In the example of Lenet, inputs are normalized 28x28 single channels, and the output of the network is a the most probable digit between 0 and 9 that this image represents. Formally described, given an input domain X, an output set of labels Y, and a set of candidate functions $f : X \to Y$ belonging to a hypothesis class H, we are trying to minimize the loss of mispredicting the correct class. The loss function can be expressed as :

$$L_D(f) = p(f(z) \neq h(z)) \tag{2.1}$$

where z is a sample from the dataset D, $f(z)$ is the output prediction and $h(z)$ is the true class or label. In that case training is then formalized as finding the best set of parameters w in our model to minimize this loss function.

$$w* = \arg\min_{w \in H} L_D(f_w) \tag{2.2}$$

Several options exists for sample loss functions as shown in Table 2.1. In regression, a common loss function is the squared error loss. In classification problems, the binary loss can be used. However, for minimization, the loss function should be both continuous and differentiable. To solve this problem, an alternative cross-entropy loss function is introduced for multi-class classification problems as opposed to the binary loss. The output becomes a probability distribution of the input according to the given possible output classes. The cross entropy loss calculates the difference between the predicted distribution and the true distribution into K classes.

Squared loss	$l = (f_w(z) - h(z))^2$
Binary loss	$l = \begin{cases} 0, & \text{if } f_w(z) = h(z) \\ 1, & \text{if } f_w(z) \neq h(z) \end{cases}$
Cross-entropy loss	$l = \sum_{i=1}^{K} f_w(z_i) \log(h(z_i))$

TABLE 2.1: Examples of loss functions

Gradient Descent

To solve the minimization problem, we can either use evolutionary algorithms inspired by natural processes and meta-heuristics or we can resort to the use of iterative approaches. It is more popular in machine learning to use the iterative gradient descent techniques. In gradient descent, starting from an initial set of weights, we calculate the gradient of the loss function with respect to the weights $\frac{\partial L}{\partial w}$ and iteratively adjust the weights in the direction of the gradient to minimize loss (η is the learning rate.

$$w^t = w^{t-1} + \eta \frac{\partial L}{\partial w} \tag{2.3}$$

The error is back-propagated layer by layer from the output to the input layer by utilizing the chain-rule.

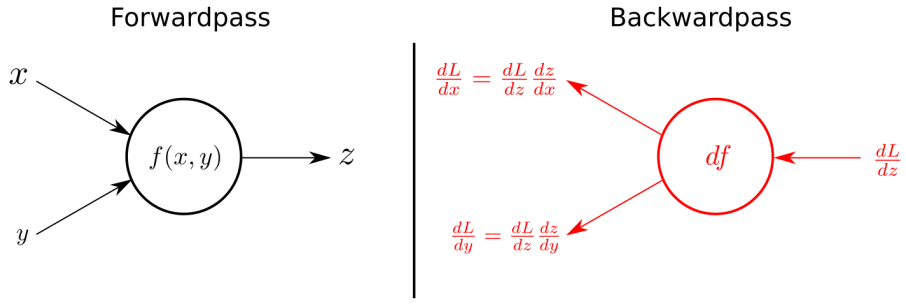

FIGURE 2.5: Local Gradient Backpropagation using the Chain Rule
Source:[1]

Gradient descent implementations vary depending on how many samples are used to calculate the error. The three main types are batch, stochastic, and mini-batch gradient descent [36]. The different types also exhibit tradeoffs in terms of computations required, frequency of updates, convergence, and stability of the calculated gradient.

Algorithm 1: Batch Gradient Descent

N = number of training points;
while *not converged* **do**
  $w^t = w^{t-1} + \eta \sum_{i=0}^{i=N} \frac{\partial L}{\partial x_i} \frac{\partial x_i}{\partial w}$;
end

In Batch Gradient Descent (BGD), the error is calculated for all of the samples in the training set. After all of the samples have been observed, the error can be backward propagated through the layers and the weights are updated. This complete forward pass and the backward update is called an epoch and usually neural networks are given a fixed set of epochs to train or we keep training until a certain accuracy is achieved. The benefit of using batch gradient descent is that the complete gradient is computationally efficient and presents stable convergence. Stability on the other hand makes it harder to avoid local minima and the minimization problem can easily get stuck in a local minimum. Knowing that the optimization problem of neural networks is riddled with saddle points, we might want to resort to different types of gradient descent.

Algorithm 2: Stochastic Gradient Descent

N = number of training points;
Randomly Shuffle Data Points;
for *i=1, ..., N* **do**
  $z \leftarrow x_i$;
  $w^t = w^{t-1} + \eta \frac{\partial L}{\partial z} \frac{\partial z}{\partial w}$;
end

Stochastic gradient descent (SGD) offers an alternative to calculating the full gradient and passing in the whole dataset. Instead, a random sample is selected from the dataset and forward propagated, the weights are then updated after backpropagating the gradient resulting from this sample. This update technique is less stable than batch gradient descent but it helps in avoiding local minima. It has also been

[1] https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalizatic
html Last Accessed: 23/08/2018

proven that it converges with a rate of $\frac{1}{\sqrt{T}}$ where t is the iteration number or epoch for convex functions. SGD demands more computational power due to the frequent updates especially when the training set is large.

Algorithm 3: Mini-batch Gradient Descent

N = number of training points;
B = batch size;
for *i=1, 1+B, ...N-B+1* **do**
  $w^t = w^{t-1} + \eta \sum_{j=i}^{j=i+B-1} \frac{\partial L}{\partial x_j} \frac{\partial x_j}{\partial w}$;
end

Minibatch gradient descent comes as the middle ground between SGD and BGD and is commonly used in practice. The weight update is viewed only after a mini-batch is forward propagated. Typical batch sizes are 32, 64, 128 (sometimes much larger in the thousands) as powers of 2 fit the memory requirements of GPU accelerators and memory requirements. This technique balances the robustness of SGD and the stability of BGD. Minibatch introduce another hyperparameter that increases design space and allows for trying out different values to obtain the best test-accuracy.

2.4 Parallelism in Deep Neural Networks

As the models grow in size and have more trainable parameters, more data and computational resources are required to properly train the above networks. For that we can speed up the training process of DNNs by exploiting parallelism from different directions. We can distinguish three main types of parallelism [3]; data parallelism by parallelizing over the input dimension, model parallelism by tiling computations and running the same layer concurrently on separate cores, and pipeline parallelism which exploiting the pipeline structure of the networks and run the layers concurrently where one layer's output feeds into the other layer directly.

2.4.1 Data Parallelism

The structure of the input and intermediate results of convolutional neural networks as multidimensional grids allows us to use this structure and split up the input into several parts that can also be forward propagated concurrently. The training samples in a batch gradient descent algorithm can be calculated separately before the back-propagation step. The bottleneck in this approach appears when we wish to back-propagate the error and all of the errors are averaged to calculate the gradient with respect to the loss function. The paradigm is suitable for a MapReduce model and can easily be parallelized. The only obstacle to this approach is batch normalization layers where synchronization has to occur at every normalization layer.

2.4.2 Model Parallelism

In this type of parallelism, the neurons in a hidden layer are divided and computed separately. This can decrease the memory requirement for running the network if the model is partitioned but it also adds the overhead of communication between the different parts of the model. For example in an all-to-all connection in fully connected layers, the intermediate results should be shuffled across different computing

nodes and synchronized so that the next layer can be computed. Some improvements have been proposed such as adding redundant computations in fully connected layers so that less communication is required[31], but it comes at the expense of more computations. As for convolutional layers, splitting the task of calculating output feature maps across separate processes would induce an overhead of reading the input map of the previous layer multiple times and is thus impractical.

2.4.3 Pipeline Parallelism

This type of parallelism is similar in a sense to both of data and model parallelism. The multiple stages of computations and layers in a DNN can be all active at the same time in a pipeline order. The idea is that different stages can be working on different parts of the pipeline as data is ready from an earlier stage. This is specifically important to the rest of the work as this is a type of parallelism that only FPGA accelerators can benefit from. On such flexible architectures, complex dataflows and pipelines can be programmed. Some implementations have shown that forward propagation and the backward pass computations can be pipelined together [2, 9]. The challenge in this is that as deep neural networks become bigger, they may not fit within a standard FPGA resources. Two solutions can thus be proposed. On one hand, the layers can be combined together and the network can perform the computations, reprogram itself and then compute another combination of layers. Communication in that case can be done by reading and writing intermediate results to global memory. Another solution can be implemented using the OpenCL SDK for Xilinx (Intel has not yet added support for that in high level synthesis) leveraging the power of partially reconfiguring an FPGA while part of it is performing computations.

Chapter 3

Hardware Implementation

3.1 LeNet Pilot

The main contribution in this work is in the form of a framework for the development and optimization of deep neural network operators on an FPGA. We use the Lenet [28] model, knowing that it is outdated and outperformed by many other network architectures discussed before, due to its simplicity and as a pilot to test our framework. The simple network allows us to implement four different types of layers (convolution, maxpooling, fully connected layers, and softmax). We also implement the backward propagation operators for all of the above layers. We use LeNet to guide the numerous optimizations we perform on each layers separately and on the combination of the layers into a pipeline as well. We implement a modern variant of Lenet which only differs slightly from the original one. The first convolution in our model contains 10 feature maps as opposed to 6 in the original lenet2.1, the second convolution outputs 20 feature maps as opposed to 15 feature maps. This adds more trainable parameters and thus we hope to use most of the MNIST dataset for training. For the sub-sampling layers we replace the average pooling operation by the maxpooling operation. We also use ReLU activation functions as opposed to the sigmoid as it has a lighter hardware implementaion and it has shown better results than the sigmoid in practice [24]. In the following sections we go more in detail about the implementation of the operators using OpenCL.

Layer	**No. Parameters**
C1	260
C2	5,020
C5	117,720
F6	10,164
Total	133,164

TABLE 3.1: Number of Trainable Parameters for Custom LeNet Implementation

3.2 Layer Implementations

3.2.1 Convolution Layer

The fundamental operation in Convolutional Neural Networks is the convolution operation. This layer takes as input a multidimensional grid and extracts output feature maps by sliding a window of weights. The filter weights and biases are trainable by gradient descent. An output feature map pixel y_i is obtained by passing a filter of size $K_h x K_w$ over an input feature x_i with CH_{in}

$$y_i = relu(\sum_{c=0}^{c=CH_{in}} \sum_{h=0}^{h=K_H} \sum_{w=0}^{K_w} w_{i,c,h,w} * x_{c,h,w} + bias) \tag{3.1}$$

The nested loop structure lends itself easily for parallelization. We will discuss three different implementations for this layer and compare tradeoffs that can be used by the user

Simple Implementation with Unrolling

With the help of high level synthesis, we can write a C-like implementation and analyze the performance. Assuming the kernel is pipelined we will require $batchsize * Img_h * Img_w * K_h * K_w * CH_{in} * CH_{out}$ cycles theoretically to complete the convolution. To speed up the naive implementation we perform some optimizations:

- **Unroll over filter computation**: Calculating each pixel in the output feature maps requires $K_h * K_w * CH_{in}$ multiplications. As our filter sizes are small, we can unroll over the filter dimensions. By unrolling we are effectively parallelizing the algorithm by $25x$ (since in our case $K_h = 5, K_w = 5$) . However, we choose to unroll over the input channels and the width of the kernel only. We choose input channels as opposed to only the kernel dimensions due to our memory layout because we stripe the input and output feature maps by channels as the lowest rank dimension. This increases memory performance by performing aligned memory reads.
- **Buffer the coefficients**: During the forward pass, the filter coefficients, will be read multiple times. For that, we can reduce the amount of memory reads and buffer the coefficients in low-power and fast registers. This uses space on the board but allows us to access coefficients for a relatively cheaper cost.

As the summation inside for the values inside a sliding window introduces memory dependencies, we buffer intermediate results in local registers and then we perform a tree-based addition to sum up the results. We end up with an optimally pipelined implementation with an iteration index (ii) = 1 and thus the theoretical number of cycles for a convolution is expressed as

$$Latency(L) \approx ii * batchsize * Img_h * Img_w * CH_{out} * K_h \; cycles \tag{3.2}$$

```
#pragma ii 1
for(int b = 0 ; b < batch_size ; b++){//loop batch size
 for(int i =0 ; i < IMAGE_HEIGHT ; i++){ //img height
  for(int j = 0 ; j < IMAGE_WIDTH ; j++ ){ //img width
   for(int chout = 0 ; chout < CH_OUT ; chout++){ //output channels
   float val[WIDTH*HEIGHT*CH_IN];  //local storage of intermediate results
    for(int k = -PAD_H; k <= PAD_H ; k++ ){ //kernel height
     #pragma unroll
     for(int l = -PAD_W ; l <= PAD_W ; l++){ //kernel width
     #pragma unroll
      for(int z = 0 ; z < CH_IN  ; z++ ){ //input channels
       //Check if out of bounds
       if(i+k>=0 && i+k<IMAGE_HEIGHT && j+l>=0 && j+l<IMAGE_WIDTH){
        val[z][k+PAD_H][l+PAD_W] =
        coeff[chout][k+PAD_H][l+PAD_W][z]*input[b][i+k][j+l][z];
       }else{
        val[z][k+PAD_H][l+PAD_W] = 0.0;
       }
      }
     }
    }
    float final_val = 0.0;

    #pragma unroll
    for(int i =0; i < CH_IN*WIDTH*HEIGHT ; i++){
    //Aggregate partial sums <- reduction
     final_val += val[i];
    }

    final_val += bias_buff[chout];
    //Write Output
    output[b][i][j][chout] = relu(final_val);
    }//end chout loop

    }
  }
}
```

LISTING 3.1: code snippet from simple convolution

Sliding Buffer Implementation

In the simple, we buffered the coefficients in low-cost local registers to avoid having to read the coefficients multiple times from memory. We also notice that when the convolution window slides with a stride of 1 step, there will be also values in the input feature map that are re-used. For that, we implement a sliding window that contain's all previously seen values as long as they can still be useful. Our implementation is similar to what is describe by Zohouri et. al [49]. The shift-register holds the input values and multiple taps allow for accessing the desired values in the sliding window. The intel offline compile performs optimizations like replicating RAM blocks to allow for simulataneous access on the register. For that, only few compiler directives should be specified to make this implementation work. This implementations is also an example of how we can trade local storage to minimize bandwidth. We also carry over the optimization done in the simple implementation.

This implementation proves useful if data is provided in a streamlined fashion. The sliding buffer reads data sequentially and is more suitable for non-blocking dataflow computations.

Limitations

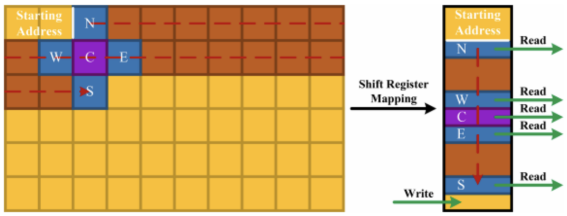

FIGURE 3.1: Sliding Buffer Visualization. Source: [49]

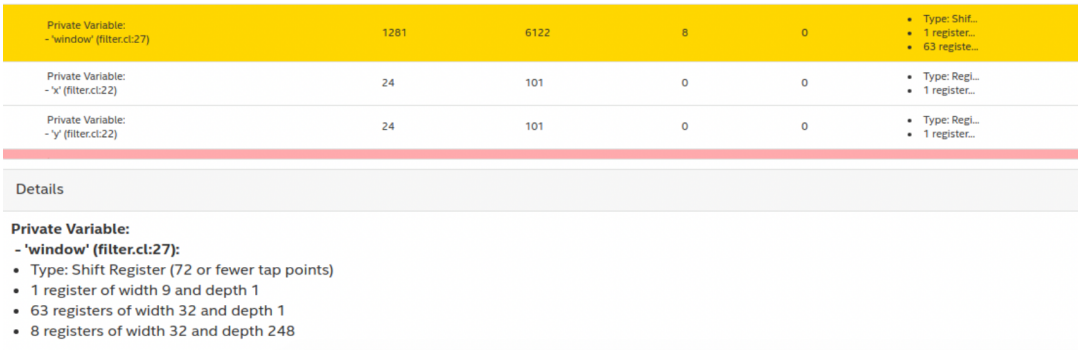

FIGURE 3.2: Sliding 'window' variable is inferred as a shift-register by the OpenCL compiler

One of the limitations this implementation is that we quickly run out of buffer space as the dimensions of the problem grow larger. The sliding buffer size grows linearly with the above parameters: CH_{in}, Img_w, K_h, K_w. For our application in a network as small as the Lenet, we do not worry about this problem. The solution to scaling the sliding buffer implementation is to utilize spatial blocking and tile the input into blocks and perform the computations in these tiles separately [49].

Row-stationary Implementation

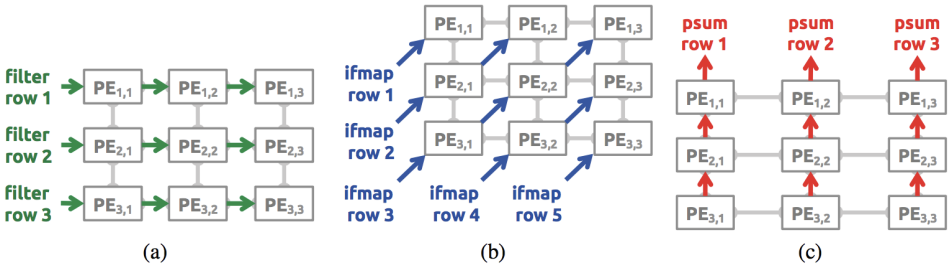

FIGURE 3.3: Datar re-use across processing elements in Eyeriss. Source: [6]

This approach was suggested by Chen. et. al [6] and it presents a way of reusing both filter weights and input feature maps. The technique requires a set of replicated

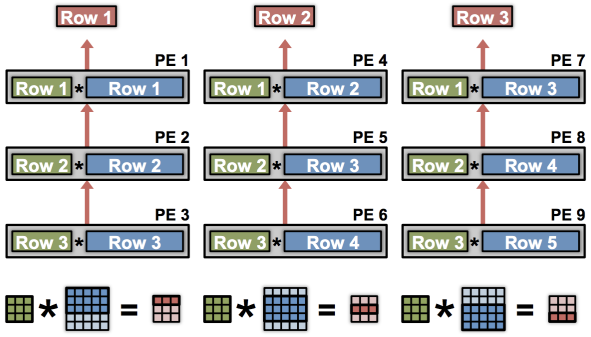

FIGURE 3.4: 2D spatial convolution with a grid of processing elements (Eyeriss). Source: [41]

processing unit. In the simple case of 3x3 filters we can instantiate 9 processing elements (PEs) each holding one of the 3x3 filter weights 3.3. We implemented a simplified version of this architecture using the Intel SDK's autorun kernels, which means the kernels do not need to be explicitly invoked and can communicate data to each other using channels (FIFOs). Input feature maps are then streamed diagonally (blue 3.3) where they are required for the computation of different output feature map rows. The partial sums are accumulated upwards vertically (shown in red 3.3) upwards.

Advantages of using this is that the size of this grid can be adjusted to best fit the resources available on a specific board. This dataflow performs the theoretical minimum of memory reads required as input fmaps are read once and communicated diagonally to other PUs based on demand. Communication between PEs is done through Intel OpenCL channels which are Intel's implmentation of FIFOs (or pipes in OpenCL terms) and computations are performed asynchronously. Two separate reader and writer kernels handle reading and writing data between global memory and the processing grid. The size of this grid is reconfigurable and the user can instantiate a larger version of this grid. It is independant of the image and filter sizes as we reuse techniques from Eyeriss [6] such as folding and replication to map different computations onto the same fixed grid.

Other Implementations

The below methods were **not implemented** but it is worth discussing other approaches to performing a convolution. The shared idea is to transform the convolution operation into another form of computation with different properties thus allowing to perform different types of optimizations.

Matrix Multiplication: One solution to the convolution problem involves transforming the input matrix and incorporating redundant data as in [7]. This transforms the convolution operation into a direct matrix multiplication. The downside of using this is the requirement of either a larger storage for storing the redundant inputs or a very complicated memory access pattern to be implemented [3].

Fast-Fourier Transform (FFT): Another solution is found by transitioning into the Fourier domain and applying a fourier transform on both the filter and the input

image [43]. In the Fourier domain, a convolution becomes again a matrix multiplication. After multiplication of the frequency domain representations of the filter and the input, we use the inverse-fourier transform to obtain the output image. The fourier-transform decreases the complexity of the convolution operation from $O(N_o^2 N_f^2)$ to $O(N_o^2 log_2(N_o)$ where the input image size is $N_o x N_o$ and the filter size is $N_f x N_f$ .The tradeoff in this case is less operations on the expense of additional bandwidth and storage requirements. The quality of this transformation and benefits degrade for smaller filters and thus fourier is usally used in the case when large input features and filters are required [41].

Strassen's Algorithm[11]: can reduce the number of multiplications from $O(N^3)$ to $O(N^{2.8})$[41] . The reduced multiplications lower the space requirement as floating point operators require more space than additions but comes at the expense of numerical stability and storage space requirement to hold and propagate intermediate results [41].

3.2.2 Maxpool Layer

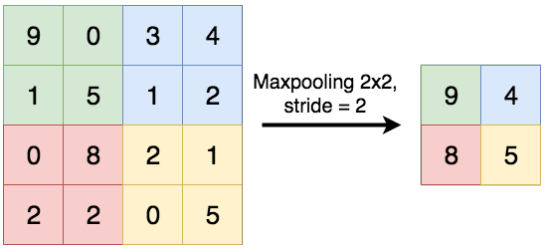

FIGURE 3.5: Maxpooling Operation

Convolution layers are usually followed by subsampling layers such as average pooling and maxpooling [28]. These operations play a main role in decreasing the size of feature maps in the network and also add to the robustness and also make the network slightly shift-invariant and robust against distortions in the image [24]. Two common subsampling operations are the average pooling where the input values within a certain window are averaged and passed as one output feature map. The second type, which we choose to implement is the maxpooling which passes only the maximum value within a certain window in the input feature map to the output feature map. We developed two different implmentations for the maxpool operation which are the bandiwdth heavy, and streamlined version.

Bandwidth heavy: the max operation within a certain window can easily be parallelized. In fact the maximum of the four values can be calculated in a single cycle and we can compute the one output pixel in the output feature map at a single clock cycle. Assuming we have sufficient bandwidth we can in fact perform the maxpool reduction to the whole input feature in one cycle however this requires reading the full feature map from memory at once which is impractical. For that, we limit the parallelization to the size of the window. In lenet, it is $2x2$ which gives us a $4x$ speedup over the naive implementation of the sequential version of the problem.

Streaming calculation: As we aim for efficiently pipelining the operators in a convolutional neural networks. We take into consideration the out input feature maps will be streamed in as an input. Therefore to perform the maxpool operation we should buffer in the whole first row and then we can proceed to produce the outputs sequentially. So assuming we are reading our data through an OpenCL

channel, the way to fix this is to borrow from the already implemented solution for the sliding buffer convolution. In fact, the maxpool operation iterates over an ifmap as a convolution however in our case the stride size is equal to the filter dimensions 2x2. We adapt the sliding buffer implementation from the convolution layer to the maxpool layer and use it in our generic maxpool template.

3.2.3 Non-Linearities

In between convolution layers and fully connected layers, non-linear functions are applied on the output feature maps. Without non-linear functions, a neural network (no matter how many layers it consists of) would behave as a single layer perceptron because summing the layers would give just another linear function. We implement both classical non-linearities such as the sigmoid function and the hyperbolic tangent, in addition to the ReLU function. The ReLU non-linearity define as $relu(x) = max(0, x)$ has proven to increase accuracy [24], in addition to accelerating training as the derivative can be simply coded as $relu'(x) = 1$ if $(x \geq 0)$ else 0 . In terms of hardware the relu derivative is simply a MUX wired to constant values of 0 and 1. We implement these operations as a streamlined operation which is pipelined with an *ii* of 1.

3.2.4 Softmax

For the output layer we implement the *Softmax* operation. It can be used as a modular output layer for a network that performs multi-class classification such as the LeNet. The operation is defined as follows:

$$f(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \qquad \textit{for j=1,...,K} \tag{3.3}$$

and it is used to map K outputs to K possible probability values in a categorical distribution. This function highlights the maximum values and suppresses those that are a certain order of magnitude below the maximum value. Due to the interdependency for normalizing the outputs, the softmax needs to read all output values before it operates, however the operation is unrolled, and since this is the final layer the output of this operation is written directly to memory.

3.2.5 Backpropagation

Backpropagation for all of the layers in the above sections. We summarize all backpropagation kernels in one section for two reasons; the first being that backpropagation operators borrow the same implementation from the forward calculations.i.e backprop of a convolutional layer is also a convolution, and backprop of a fully connected layer is also a matrix multiplication. We believe that the main challenge introduced by implementing backpropagation is that It requires additional buffer space to hold intermediate outputs. For that we are faced with two options:

- Replicate writes in the pipeline to global memory and find a way to synchronize forward and backward operations.
- Use kernels that read and write to memory and run them sequentially and then eventually run backpropagation.

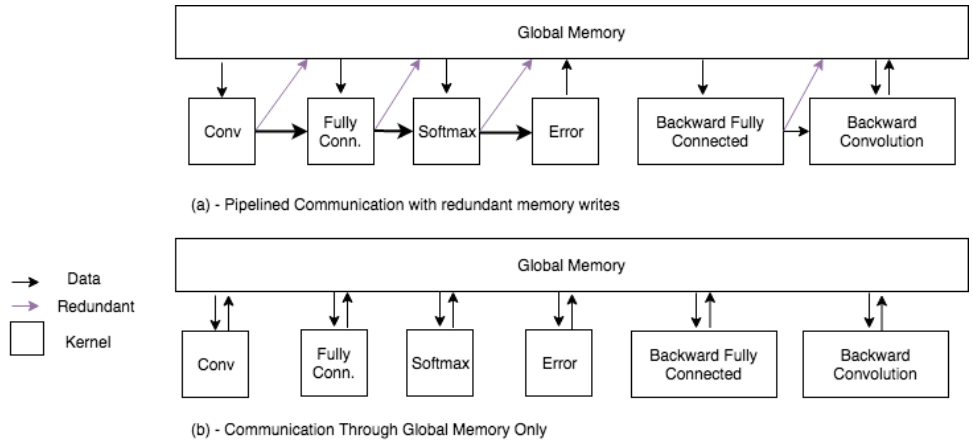

FIGURE 3.6: Two types of inter-layer communication

We choose to go with option (b) in training neural networks, though not optimal, due to time limit constraints. Our backpropagation algorithm is not pipelined only due to the fact that we need to store intermediate results and this makes it easier as a first prototype for training neural networks on FPGAs. This only impacts performance slightly, because when kernels communicate through memory several parallelization techniques can be performed while sacrificing bandwidth. Also as each kernel runs in a separate timeslot, the whole board bandwidth is available for the single layer which offers us more flexibility in parallelization and making use of the maximum bandwidth. Another thing to note is that profiling kernels is easier and we can better see the time consumed by each kernel and relieve bottlenecks for different operators or architectures

Backpropagation is the process of propagating an an error back to the network layers in order to adjust the weights. The gradient is propagated backwards in the network based on the chain rule in differential calculus. Several weight update rules ranging from a fixed learning rate to adaptive and momentum-based methods are explored in literature [3]. In our implementation we keep the learning rate as an input argument so that it can be configured at runtime by the host program. This allows for flexibility in experimenting with different weight update rules without the need to rebuild the kernel.

The trainable weights are usually initialized to small random values in the range $(-0.5, 0.5)$. In practice, the random weights are then normalized depending on the weight activations used[1]. There are also different weight normalization schemes used in practice to mitigate (but not completely solve) problems that arise in deeper networks like the vanishing or exploding gradient [18]. For the ReLu activation, it is common to multiply the random weights by a factor of $\sqrt{\frac{2}{sizeof(prev.layer)}}$. This initialization is done on the host side and the weights are then transferred to the FPGA in the beginning of the training procedure.

Convolution Backpropagation: The backpropagation of a convolution operator is also a convolution. For that we borrow some optimizations performed on the forward convolution such as buffering the coefficients, and unrolling over kernel dimensions. We note the main difference is that in inference phase, a window of dimensions $K_w x K_h$ is sliding along an input feature map in order to obtain an output

[1] https://medium.com/usf-msds/deep-learning-best-practices-1-weight-initialization-14e5c0295b94 Last Accessed: 25/08/2018

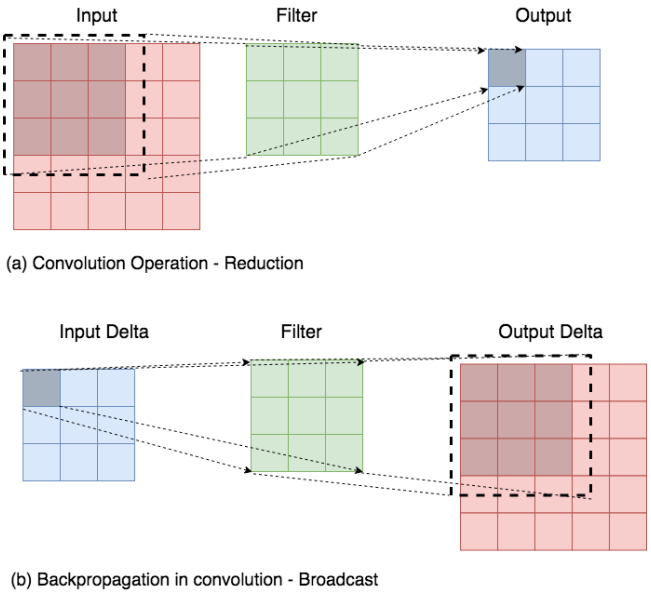

FIGURE 3.7: Propagation of data in convolution

feature map. Thus this is more of a *reduction* performed by multiply and accumulation of values in a specific window. In the case of propagating back an error, the input delta (coming from the output feature map) is *broadcast* to all of the pixels in the input feature maps (also show a picture here). A simple implementation would introduce memory dependencies on calculations, another way is to rearrange dimensions and use local buffering to hold values of the output deltas before writing to memory. The backpropagation kernel we implemented is fully pipelined with $ii = 1$ and also performs batched weight updates for the filters.

Maxpool backpropagation: For backpropagation through maxpool, we simply pass upstream the error to the input pixel in a certain window with the maximum value. The same implementation from the forward max pooling is used with the addition of passing an output gradient at the maximum as opposed to passing the maximum value downstream. In our case the maxpool does not have any trainable parameter and acts as a multiplexor that passes down the error as it is.

Fully connected backpropagation: The fully connected layers used for classification contain most of our trainable parameters. In the case of minibatches we use a matrix multiplication technique for both forward and backward propagation of the error. We also try to unroll computation loops to utilize the maximum bandwidth available for this kernel. The backpropagation is a similar inverse computation (still matrix multiplication) and the weights are also updated according to a configurable learning rate.

Loss: For the loss function we implemented a cross entropy loss function (2.1). It measures divergence of the softmax output probabilities (representing a categorical distribution) with the target output probabilities as a *one-hot* encoding.

3.3 OpenCL Kernel Template Generator

Many parameters such as sliding window sizes, kernel size, and number of input and output channels should be known at compile time. Each of the discussed layers is parametrized by a specific set of parameters that it takes in as a configuration. For

that we created a python tool that instantiates instances given a kernel template in '.cl' format (the extension for OpenCL kernels). The idea is to use a configuration file in JSON to customize a specific layer. In conjunction, we used a python library called jinja2[2] to instantiate templates.Templating allows us to instantiate multiple versions of the kernel to be used and compiled together for the neural network accelerator.

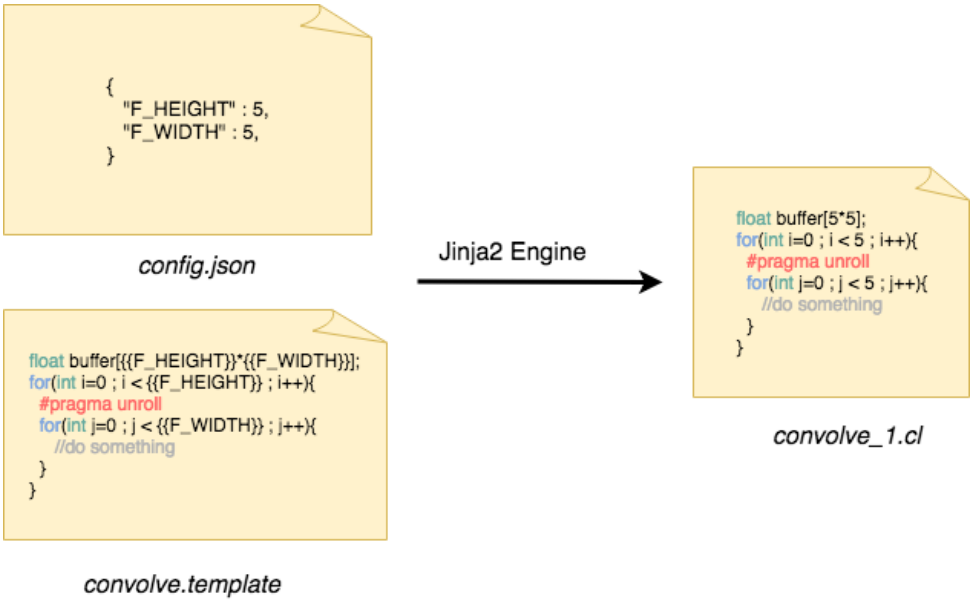

FIGURE 3.8: Templating Kernels

```
hnaous@greina2:~/dnn_fpga$ ./script.py -h
usage: script.py [-h] [-f] [-p] {cpp,emulate,report,hw} filename context

positional arguments:
  {cpp,emulate,report,hw}
                        enter a command
  filename              template file location
  context               JSON configuration file for template

optional arguments:
  -h, --help            show this help message and exit
  -f, --fast_compile    enable fast-compile
  -p, --profile         enable profiling
```

FIGURE 3.9: OpenCL Template Generator Usage

The basic usage is exposed through the interface shown in 3.9. The Python tool allows the user to also perform additional tasks on the generated kernel such as generating the performance report, compiling for emulation, compiling for hardware, and compiling the kernel as a as a C++ application using gcc. We discuss the use of compiling as a C++ application in section 5(REF HERE). This type of compilation differs from the traditional workflow suggested by Intel as we have extended the emulation framework of kernels and use C++ to perform unit testing and verifying correctness of the operators. This rids us from the responsibility of creating a host program that interfaces with the kernel and from a lot of overhead code in managing the device buffers just for the sake of verification. As for the additional functions

[2] http://jinja.pocoo.org/docs/2.10/ Last Accessed: 25/08/2018

such as generating report, and compiling for hardware, the additional value provided by the tool is that it supports instantiating multiple templates at the same time and combining them in a single report or a single binary. All of the kernel implementations discussed in 3 are designed as templates and the generator tool is used for instantiation, and interaction with the Intel compiler to perform different types of analysis.

3.4 Integration with Deep500

Deep500[3] is a library developed internally in the Scalable and Parallel Computing Lab at ETH to aid in the process of developing custom backends for computational graphs. It enables the extension of operators, verification, network optimization, and acts as a distributed learning framework for deep neural network models. The networks are expressed in ONNX[4] format. ONNX is an open source format to represent computational graphs (specifically deep neural networks). It allows the interoperability between different deep learning frameworks such as torch[8] and tensorflow[1] and the exchange of models and parameters in between those frameworks for performing optimizations, training, and inference.

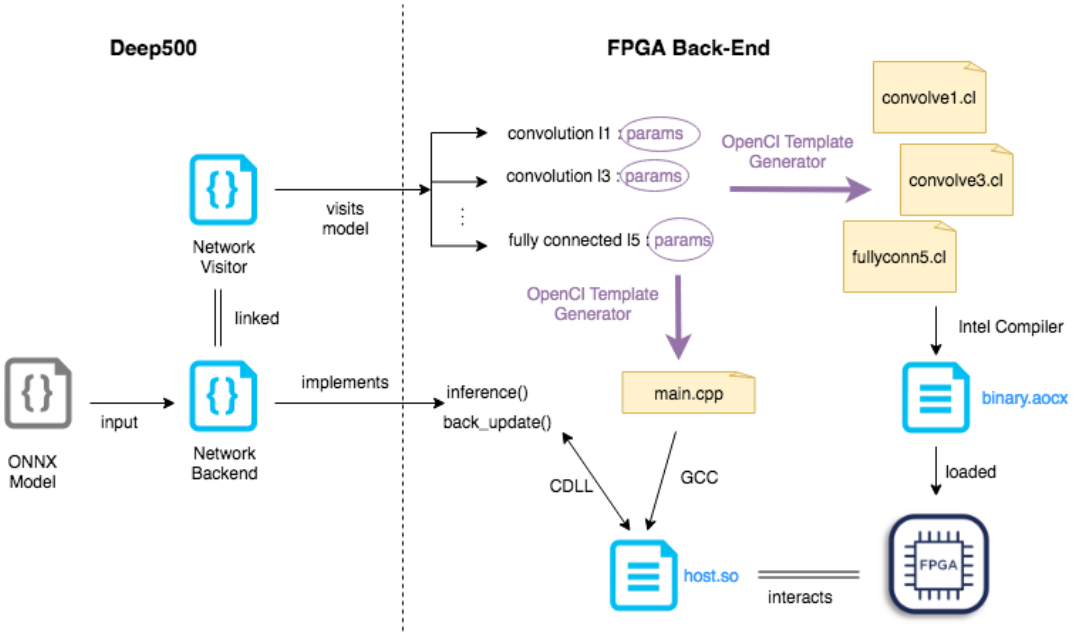

FIGURE 3.10: Custom FPGA Back-End for ONNX Models

The Deep500 library provides us with an extendable visitor interface to implement each of the operators required for a given onnx model. For example in our network, the operations we implement are convolution, maxpooling, relu, and gemm (generalized matrix multiplication). The Deep500 library traverses the model's operations using a NetworkVisitor class. We extend the visitor class to implement all of the mentioned operations. Each visit to a node in the graph provides the parameters and input shapes required for implementing this node's operation. We use those parameters to instantiate an OpenCL kernel using the template generator we created.

[3] https://github.com/deep500 Last Accessed 25/08/2018
[4] https://github.com/onnx/onnx Last Accessed : 25/08/2018

With the *.cl* files available, we are able to transform an ONNX model into a group of inter-related OpenCL kernels that can be built for hardware 3.10. From there we can use again the tool to view the area utilization report, emulate the kernel, and perform additional unit tests on each of the operations. To enable training and inference using the kernels and built CL files, we require a host program running on the CPU that interacts with the above kernels. The host program is in C++ and has to be customized for the computational graph it runs. For that we choose to template the host program also using *jinja2*. In addition, we also use the operation parameters to be able to generate a suitable host program. *jinja2* allows us to use nested JSON configurations and arrays in order to generate the host program. The host program performs the following operations : setting up opencl environment, passing data between the host program and the FPGA's global memory buffer, and enqueuing kernels to be executed on the FPGA.

For the accelerator backend to work it should communicate with the templated host program, so after instantiating the host program, we compile it as a shared object *'host.so'*. This allows us to use python's CDLL library to create a handle to the C++ program. The forward and backward propagation functions are exposed externally to CDLL and we can easily pass down coefficients and inputs to the host program,which executes and returns the results.

The added value and end result of this work is given an ONNX model we can create and compile the relevant FPGA binaries by using Deep500 to traverse the graph. We are also able to instantiate and compile a controller host program, and interact with this host program to train a neural network. The benefit also extends to other popular DNN frameworks where models can be exported to onnx. For example the pytorch library offers the option of exporting torch models to onnx, the implementation offers the ability to interact with this tool and train other networks on an FPGA. It is also worth mentioning that Deep500 allows us to perform unit-tests to verify each of the implemented operations on the FPGA.

Appendix A

Frequently Asked Questions

A.1 How do I change the colors of links?

The color of links can be changed to your liking using:

`\hypersetup{urlcolor=red}`, or
`\hypersetup{citecolor=green}`, or
`\hypersetup{allcolor=blue}`.

If you want to completely hide the links, you can use:

`\hypersetup{allcolors=.}`, or even better:
`\hypersetup{hidelinks}`.

If you want to have obvious links in the PDF but not the printed text, use:

`\hypersetup{colorlinks=false}`.

Bibliography

[1] Martín Abadi et al. "Tensorflow: a system for large-scale machine learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[2] Martín Abadi et al. "Tensorflow: a system for large-scale machine learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[3] Tal Ben-Nun and Torsten Hoefler. "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis". In: *arXiv preprint arXiv:1802.09941* (2018).

[4] James Bergstra et al. "Theano: Deep learning on gpus with python". In: *NIPS 2011, BigLearning Workshop, Granada, Spain*. Vol. 3. Citeseer. 2011, pp. 1–48.

[5] Sheng Chen, Colin FN Cowan, and Peter M Grant. "Orthogonal least squares learning algorithm for radial basis function networks". In: *IEEE Transactions on neural networks* 2.2 (1991), pp. 302–309.

[6] Yu-Hsin Chen et al. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks". In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138.

[7] Sharan Chetlur et al. "cuDNN: Efficient Primitives for Deep Learning". In: *CoRR* abs/1410.0759 (2014). arXiv: 1410.0759. URL: http://arxiv.org/abs/1410.0759.

[8] Ronan Collobert, Samy Bengio, and Johnny Marithoz. *Torch: A Modular Machine Learning Software Library*. 2002.

[9] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. "Torch7: A matlab-like environment for machine learning". In: *BigLearn, NIPS workshop*. EPFL-CONF-192376. 2011.

[10] Katherine Compton and Scott Hauck. "Reconfigurable computing: a survey of systems and software". In: *ACM Computing Surveys (csuR)* 34.2 (2002), pp. 171–210.

[11] Jason Cong and Bingjun Xiao. "Minimizing computation in convolutional neural networks". In: *International conference on artificial neural networks*. Springer. 2014, pp. 281–290.

[12] Balázs Csanád Csáji. "Approximation with artificial neural networks". In: *Faculty of Sciences, Etvs Lornd University, Hungary* 24 (2001), p. 48.

[13] C Cuda. *Programming guide*. 2012.

[14] Jeffrey Dean et al. "Large scale distributed deep networks". In: *Advances in neural information processing systems*. 2012, pp. 1223–1231.

[15] Roberto DiCecco et al. "Caffeinated FPGAs: FPGA framework for convolutional neural networks". In: *Field-Programmable Technology (FPT), 2016 International Conference on*. IEEE. 2016, pp. 265–268.

[16] Rajesh Gupta and Forrest Brewer. "High-Level Synthesis: A Retrospective". In: *High-Level Synthesis: From Algorithm to Digital Circuit*. Ed. by Philippe Coussy and Adam Morawiec. Dordrecht: Springer Netherlands, 2008, pp. 13–28. ISBN: 978-1-4020-8588-8. DOI: 10.1007/978-1-4020-8588-8_2. URL: https://doi.org/10.1007/978-1-4020-8588-8_2.

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

[18] Sepp Hochreiter. "The vanishing gradient problem during learning recurrent neural nets and problem solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116.

[19] Itay Hubara et al. "Binarized neural networks". In: *Advances in neural information processing systems*. 2016, pp. 4107–4115.

[20] David H Hubel and Torsten N Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". In: *The Journal of physiology* 160.1 (1962), pp. 106–154.

[21] Forrest Iandola et al. "Densenet: Implementing efficient convnet descriptor pyramids". In: *arXiv preprint arXiv:1404.1869* (2014).

[22] FPGA Intel. "SDK for OpenCL". In: *Programming Guide. UG-OCL002* 31 (2016).

[23] Yangqing Jia et al. "Caffe: Convolutional architecture for fast feature embedding". In: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 2014, pp. 675–678.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[25] Griffin Lacey, Graham W Taylor, and Shawki Areibi. "Deep learning on fpgas: Past, present, and future". In: *arXiv preprint arXiv:1602.04283* (2016).

[26] Andrew Lavin and Scott Gray. "Fast algorithms for convolutional neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 4013–4021.

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), p. 436.

[28] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[29] Grant Martin and Gary Smith. "High-level synthesis: Past, present, and future". In: *IEEE Design & Test of Computers* 26.4 (2009), pp. 18–25.

[30] John McCarthy et al. "A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955". In: *AI magazine* 27.4 (2006), p. 12.

[31] UA Muller and A Gunzinger. "Neural net simulation on parallel computers". In: *Neural Networks, 1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on*. Vol. 6. IEEE. 1994, pp. 3961–3966.

[32] Eriko Nurvitadhi et al. "Can fpgas beat gpus in accelerating next-generation deep neural networks?" In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2017, pp. 5–14.

[33] Yohhan Pao. "Adaptive pattern recognition and neural networks". In: (1989).

[34] Rajat Raina, Anand Madhavan, and Andrew Y Ng. "Large-scale deep unsupervised learning using graphics processors". In: *Proceedings of the 26th annual international conference on machine learning*. ACM. 2009, pp. 873–880.

[35] Frank Rosenblatt. *The Perceptron : a theory of statistical separability in cognitive systems*. eng. United States Department of Commerce, 1958.

[36] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[37] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), p. 533.

[38] Suhap Sahin, Yasar Becerikli, and Suleyman Yazici. "Neural network implementation in hardware using FPGAs". In: *International Conference on Neural Information Processing*. Springer. 2006, pp. 1105–1112.

[39] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[40] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.3 (2010), pp. 66–73.

[41] Vivienne Sze et al. "Efficient processing of deep neural networks: A tutorial and survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[42] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[43] Nicolas Vasilache et al. "Fast convolutional nets with fbfft: A GPU performance evaluation". In: *arXiv preprint arXiv:1412.7580* (2014).

[44] Dong Wang, Ke Xu, and Diankun Jiang. "PipeCNN: an OpenCL-based open-source FPGA accelerator for convolution neural networks". In: *Field Programmable Technology (ICFPT), 2017 International Conference on*. IEEE. 2017, pp. 279–282.

[45] Kaining Wang and Anthony N Michel. "Robustness and perturbation analysis of a class of artificial neural networks". In: *Neural networks* 7.2 (1994), pp. 251–259.

[46] Bernard Widrow and Michael A Lehr. "30 years of adaptive neural networks: perceptron, madaline, and backpropagation". In: *Proceedings of the IEEE* 78.9 (1990), pp. 1415–1442.

[47] Peter Wilson. *Design Recipes for FPGAs: Using Verilog and VHDL*. Newnes, 2015.

[48] AXI Xilinx Vivado. *Reference Guide*.

[49] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. "Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2018, pp. 153–162.